Aspektorientierung und Programmfamilien im Betriebssystembau

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Olaf Spinczyk

geb. am: 6. April 1970 in Berlin

Gutachter:

Prof. Dr. Wolfgang Schröder-Preikschat
Prof. Dr. Reiner Dumke
Prof. Dr. Franz Hauck

Promotionskolloquium:

Magdeburg, den 16. Dezember 2002

Zusammenfassung

Olaf Spinczyk: **Aspektorientierung und Programmfamilien im Betriebssystembau**

Beim Bau maßgeschneiderter Betriebssystemsoftware, wie sie zum Beispiel im Bereich kleinster eingebetteter Systeme benötigt wird, spielt das Konzept der **Programmfamilie** eine wichtige Rolle. Damit werden die Gemeinsamkeiten und Unterschiede verschiedener Varianten eines Systems modelliert, so dass bei der Implementierung ein hohes Maß an Wiederverwendung erreicht wird.

An die Grenzen der Wiederverwendung stößt man jedoch, wenn der für die Implementierung einer bestimmten Systemeigenschaft benötigte Code über weite Teile des Systems verstreut vorliegen muss. Dies gilt beispielsweise für Synchronisationscode. Es handelt sich hierbei um sogenannte ***Crosscutting Concerns***. Eine Lösung für dieses Problem verspricht die **aspektorientierte Programmierung (AOP)**. Sie erlaubt, *Crosscutting Concerns* modular in Form sogenannter **Aspekte** zu implementieren.

Bisher war die Realisierung eines aspektorientierten Betriebssystems mangels sprachlicher Unterstützung praktisch nicht möglich, obwohl AOP gerade im Kontext von Betriebssystemfamilien sehr vielversprechend ist. So könnten strategische Entwurfsentscheidungen wie die Granularität der Unterbrechungssynchronisation oder die Zuordnung von Kontrollfäden zu Funktionen im Systemkern anders als bisher modular umgesetzt werden, so dass sie in Abhängigkeit vom Anwendungsszenario konfiguriert werden könnten.

Das Ziel dieser Arbeit bestand daher darin, alle nötigen Voraussetzungen zu schaffen, um mit konkreten Fallstudien Erfahrungen mit diesem Ansatz zu sammeln. Dazu wurde neben einer aspektorientierten Erweiterung des Entwurfsmodells der funktionalen Hierarchie eine ganze Suite sogenannter Aspektweber entwickelt. Einer davon ist der Übersetzer für **AspectC++**, einer Spracherweiterung für C++, mit der Aspekte sehr ressourcensparend implementiert werden können.

Mit Hilfe von Fallstudien im Kontext der Betriebssystemfamilie **PURE** wird in dieser Arbeit gezeigt, dass die erhofften Vorteile des Ansatzes tatsächlich eintreten. So bestimmen in einer Studie wenige Aspekte die Architektur und das Synchronisationsschema von Gerätetreibern. In einer anderen Studie wird die Strategie zur Unterbrechungssynchronisation, die sich bisher auf 15 Klassen auswirkte, durch nur einen Aspekt ausdrückt.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die in den letzten Jahren meine Arbeit unterstützt haben, so dass ich nun meine Zeit an der Otto-von-Guericke-Universität Magdeburg mit dieser Dissertation abrunden kann.

In erster Linie gilt diese Anerkennung meiner Frau Ute. Ihre tatkräftige Unterstützung und ihr Lob haben mir – gerade wegen ihres großen Sachverstandes – sehr geholfen. Außerdem hat sie zusammen mit meinen Söhnen Niklas und Simon, meinen Eltern und meinen Schwiegereltern dafür gesorgt, dass ich mich in den letzten Monaten zu 100 Prozent auf diese Arbeit konzentrieren konnte.

Ein weiterer Dank geht an die Gutachter, die die Mühe haben, diese fast 200 Seiten lange Arbeit zu lesen und eine Note zu finden. Dies ist allein schon zeitlich eine ernome Belastung und ich hoffe, dass der Inhalt sie ein wenig entschädigt.

Prof. Schröder-Preikschat bin ich besonders verpflichtet. Er hat diese Arbeit betreut und mir dabei die Freiheit gelassen, in einer Arbeitsgruppe für Betriebssysteme und Verteilte Systeme viel Zeit mit Sprachkonzepten, Übersetzerbau und Werkzeugen zu verbringen. Obwohl auch noch andere meinem Beispiel gefolgt sind, konnte ich mir seiner Unterstützung immer sicher sein.

Darüber hinaus danke ich meinen Mitstreitern im AspectC++-Team Daniel Mahrenholz, Andreas Gal sowie den inzwischen zahlreichen Anwendern. Durch Andreas entschlossene Art ist dieses Projekt überhaupt erst in Angriff genommen worden und Daniel hat durch seine Implementierungsarbeiten und viele Diskussionen stark zum Gelingen beigetragen.

Im Laufe der Jahre standen mir auch viele Studenten tatkräftig zur Seite, allen voran Matthias Urban. Er hat über Jahre kontinuierlich zusammen mit mir in einer perfekten Teamarbeit an PUMA gearbeitet und so wesentlich zur Umsetzung meiner Ideen beigetragen. Daneben sind hier noch Mario Friedrich, Henry Jesuiter, Michael Schulze, André Maaß und Sven Apel zu nennen, die sich alle wie Daniel Mahrenholz von mir haben überzeugen lassen, eine Studien- oder Diplomarbeit im Kontext meines Themenbereichs anzufertigen.

Zu guter Letzt möchte ich noch der ganzen Arbeitsgruppe, die dazu beigetragen hat, dass die vergangenen Jahre in Magdeburg trotz viel Arbeit eigentlich immer Spaß gemacht haben, und auch den Geschäftsführern der pure-systems GmbH, Danilo Beuche und Holger Papajewski, für ihr Vertrauen in meine Ideen danken.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Zielsetzung der Arbeit 3
1.3 Gliederung 3

2 Grundbegriffe **7**
2.1 Objektorientierung 7
2.1.1 Vererbung 7
2.1.2 Dynamisches Binden 9
2.1.3 Entwurfsmuster 9
2.2 Programmfamilien 10
2.2.1 Grundlagen 10
2.2.2 GenVoca 12
2.2.3 Feature-basierte Konfigurierung 15
2.3 Aspektorientierte Programmierung 16
2.3.1 AspectJ 19

3 Stand der Kunst **21**
3.1 Produktlinien 21
3.2 Programmspezialisierung 22
3.3 AOP Anwendungen im Systemkontext 24
3.3.1 Der a-kernel 24
3.3.2 Die aspektorientierte Softwarearchitektur für Betriebssysteme AOSA 24
3.3.3 Das anwendungsorientierte Betriebssystem EPOS 25
3.3.4 AOP in angrenzenden Bereichen 25
3.4 Aspektsprachen zur Systemprogrammierung 26
3.4.1 FOG 26
3.4.2 AOP mit C++ *Template*-Metaprogrammierung 28
3.5 Zusammenfassung 30

4 Problemanalyse und Lösungsansatz 33
4.1 Hintergrund . . . 33
4.1.1 Wiederverwendung versus Effizienz . . . 33
4.1.2 Statische und dynamische Konfigurierungsmaßnahmen . . 35
4.1.3 Beispiel: PURE . . . 36
4.2 Problemdiskussion . . . 39
4.2.1 Konfigurationsvielfalt . . . 39
4.2.2 Crosscutting Concerns . . . 41
4.2.3 Auswahl der Implementierungsstruktur . . . 41
4.3 Lösungsansatz . . . 44
4.3.1 Aspekte im Einzelsystem . . . 45
4.3.2 Aspekte in der Programmfamilie . . . 46
4.3.3 Zu erwartende Vorteile . . . 47
4.4 Zusammenfassung . . . 48

5 AOP in Programmfamilien 51
5.1 Unterstützung des Entwurfsprozesses . . . 51
5.1.1 Feature Modelle . . . 51
5.1.2 UML Diagramme . . . 52
5.1.3 GenVoca Modell versus funktionale Hierarchie . . . 53
5.1.4 Beziehungen von Aspekten zu Komponentencode . . . 55
5.1.5 Beziehungen von Aspekten zu anderen Aspekten . . . 58
5.1.6 *Concern*-Hierarchien . . . 60
5.2 Unterstützung des Implementierungsprozesses . . . 62
5.2.1 Arten von Crosscutting Concerns . . . 62
5.2.2 Notwendige Werkzeuge oder Sprachen . . . 64
5.3 Bewertung . . . 65
5.4 Zusammenfassung . . . 65

6 PUMA 67
6.1 Anforderungen . . . 67
6.2 Probleme . . . 69
6.2.1 Analyse von C++ Code . . . 69
6.2.2 Erweiterbarkeit der Implementierung . . . 70
6.2.3 Integration des Präprozessors . . . 71
6.3 Entwurf . . . 72
6.3.1 Schichtenstruktur . . . 72
6.3.2 Die *Parser*-Familie . . . 75
6.3.3 Codeanalyse . . . 76
6.3.4 Codemanipulation . . . 77
6.3.5 Aspekte . . . 78
6.4 Implementierung . . . 79
6.5 Benutzung . . . 81
6.5.1 Die Programmierschnittstelle . . . 81

6.5.2 Visuelles Entwickeln von Transformationsnetzwerken . . 83
6.6 Zusammenfassung 84

7 AspectC++ 85
7.1 Anforderungen 85
7.2 Probleme 87
7.2.1 Übersetzung 87
7.2.2 Syntax 87
7.2.3 Semantik 88
7.2.4 Laufzeitumgebung 89
7.3 Entwurf 90
7.3.1 Überblick 90
7.3.2 *Pointcuts* und Vergleichsmuster 91
7.3.3 Aspekte, *Advice* und Einfügungen 99
7.3.4 Laufzeitunterstützung 104
7.3.5 Geplante Erweiterung 108
7.4 Implementierung 109
7.4.1 Struktur 109
7.4.2 Übersetzungseinheiten und Transformationsvorgang . . . 111
7.4.3 Codegenerierung 112
7.5 Benutzung 115
7.6 Zusammenfassung 116

8 Weitere Aspektweber 119
8.1 COMA 119
8.1.1 Generischer, flexibler oder spezifischer Programmcode . . 119
8.1.2 Offene Komponenten 122
8.1.3 Sprachliche und technische Umsetzung 124
8.1.4 Ergebnis 125
8.2 SOSP 126
8.2.1 Die Funktionseinbettungsstrategie 127
8.2.2 Struktur und Benutzung 128
8.2.3 Ergebnis 129
8.3 Zusammenfassung 133

9 Fallstudien 135
9.1 Prozesszustände 135
9.1.1 Abstrakte und konkrete Zustände 135
9.1.2 Abstrakte Prozesszustände in der Programmfamilie 137
9.1.3 Verbindung von abstrakten und konkreten Zuständen . . . 138
9.1.4 Strukturierung des *Crosscutting Concerns* "abstrakter Prozesszustand" 140
9.1.5 Ergebnis des Entwurfsprozesses 143
9.1.6 Bewertung 144

9.2 Unterbrechungssynchronisation 145
9.2.1 Der Status quo 146
9.2.2 Problemanalyse 147
9.2.3 Variante 1 149
9.2.4 Variante 2 151
9.2.5 Vergleich 151
9.2.6 Bewertung 153
9.3 Fadensynchronisation 154
9.3.1 Synchronisationsbedarf 154
9.3.2 Gegenseitiger Ausschluss 157
9.3.3 Leser-/Schreiber-Synchronisation 159
9.3.4 Nachrichtenbasierte Kommunikation zur Synchronisation 161
9.3.5 Vergleich der Varianten 163
9.3.6 Fadensynchronisationsstrategie 164
9.3.7 Bewertung 165
9.4 Zusammenfassung 167

10 Ausblick 169
10.1 Änderungen und Erweiterungen an AspectC++ 169
10.1.1 Spezifikation und Verifikation 169
10.1.2 Schutzkonzept 170
10.1.3 Erkennung von Konfigurierungsfehlern 172
10.2 Unterstützung des Entwicklungsprozesses 172
10.2.1 Methodische Unterstützung 173
10.2.2 Entwurfsmuster 173
10.2.3 Entwicklungswerkzeuge 174
10.3 Weitere Anwendungen 174
10.3.1 Vision: AspectOS 175
10.3.2 Erwartete Ergebnisse 178
10.4 Zusammenfassung 179

11 Rückblick und Diskussion 181
11.1 Geleistete Arbeiten, Erfahrungen und Pläne 181
11.2 Vorgehen und Einordnung 184

Literaturverzeichnis 187

Abbildungsverzeichnis

2.1 Vererbungsbeziehung im Klassendiagramm 8
2.2 Struktur des Strategiemusters 9
2.3 Die Funktionale Hierarchie des FAMOS Systems 11
2.4 Eine Hierarchie von OOVMs 13
2.5 Ein Feature Diagramm des Konzepts "Auto" 15
2.6 Der Vorgang des Aspektwebens 18
2.7 Modulare Implementierung eines *Crosscutting Concerns* 18

3.1 Generischer Programmcode von `memmove()` 22
3.2 Spezialisierter Programmcode von `memmove()` 23
3.3 Aspekte mit Hilfe von *Templates* 29

4.1 Konflikt zwischen Effizienz und Wiederverwendbarkeit 34
4.2 Dualität von minimalen Erweiterungen einer Programmfamilie und objektorientierter Vererbung 36
4.3 Die PURE Fadenhierarchie 38
4.4 Abhängigkeitsbeziehung zwischen Dateisystemen und Gerätetreibern 42
4.5 Dynamisches Binden von Dateisystemimplementierungen an verschiedene Blocktreiber 43
4.6 Übliche Einsatzformen für Aspekte 45
4.7 Aspekte und Konfigurierung 47
4.8 Reduktion von Konfigurationspunkten durch konfigurierbare Aspekte 48

5.1 Feature Diagramm des Konzepts Kontrollfaden 52
5.2 Verfeinerung einer funktionalen Hierarchie mit Hilfe einer GenVoca Beschreibung 54
5.3 Verfeinerung einer funktionalen Hierarchie mittels konfigurierbarer Funktionen 55
5.4 Notwendiges Einwirken aus Sicht des Komponentencodes 57
5.5 Notwendige Aktivierung aus Sicht des Aspekts 57
5.6 Beziehungen zwischen Aspekten und herkömmlichen Modulen . . 58
5.7 Aspektinteraktionen 59
5.8 Ordnungsbeziehungen von Aspekten 60

5.9 Abbildung von *Concern*-Beziehungen auf Abhängigkeitsbeziehungen im Sinne funktionaler Hierarchien 61

6.1 Die Familie der C-basierten Sprachen 70
6.2 Die Schichtenstruktur des PUMA Systems 73
6.3 Arbeitsweise der Scannerebene 73
6.4 Ein Syntaxbaum und seine Verbindungen zu *Tokens* 74
6.5 Die Struktur der *Parser*-Familie für PUMA 75
6.6 Der *Match*-Mechanismus von PUMA 77
6.7 Aspekte des PUMA *Parsers* 79
6.8 Wiederverwendung beim Übergang vom C zum C++ *Parser* . . . 81
6.9 Das PUMA *Visual Development Kit* 83

7.1 Erweiterung der C++ Grammatik für AspectC++ 91
7.2 Schematische Erläuterung des *Pointcut*-Begriffs 92
7.3 Verbindungspunkte und deren Beziehungen 95
7.4 Ein wiederverwendbarer Kontrollflussverfolgungsaspekt 106
7.5 Ein Ausschnitt aus der `ACSyntax` Klasse 110
7.6 Struktur der AspectC++ Übersetzer Implementierung 110
7.7 Potentielle Einwirkungsbeziehungen zwischen Übersetzungseinheiten 112
7.8 Beispiel für die Codegenerierung des AspectC++ Übersetzers . . . 113
7.9 Generierter Maschinencode zum Beispiel in Abb. 7.8 114
7.10 Syntax der `asm` Anweisung im GNU C/C++ Dialekt 116
7.11 Der GNU C/C++ Aspekt von PUMA 117

8.1 Eine funktionale Hierarchie und passende alternative Klassenstrukturen 120
8.2 Klassenstruktur einer Speicherverwaltungskomponente 122
8.3 Restrukturierung einer Speicherverwaltungskomponente 123
8.4 Eine XML-basierte Komponentenbeschreibung 124
8.5 Eine Strukturanforderung für eine offene Komponente 125
8.6 Benutzung des `inline` Schlüsselworts 127
8.7 Globale Einbettungsstrategien durch SOSP und Strategie-Skripte . 128
8.8 Verbesserung der Codegrößen verschiedener Systeme im Vergleich zum Durchschnitt bei Anwendung unterschiedlicher Einbettungsstrategien 131

9.1 Die wichtigsten Prozesszustände 136
9.2 Feature-Diagramm für das Konzept des abstrakten Prozesszustands 138
9.3 Beziehung zwischen konkreten und abstrakten Prozesszuständen in einer *Concern*-Hierarchie 139
9.4 Verfeinerung des *Crosscutting Concerns* “abstrakter Prozesszustand” 141
9.5 Klassen/Aspekt-Struktur des “abstrakten Prozesszustands” 143
9.6 Kritische Abschnitte im Betriebssystemkern 145

9.7 Zugriffe auf die globale Sperrvariable in PURE 147
9.8 Variante 1 des Aspekts zur Unterbrechungssynchronisation 150
9.9 Variante 2 des Aspekts zur Unterbrechungssynchronisation 152
9.10 Zugriff auf eine geteilte Betriebsmittel durch Fäden 155
9.11 Unerwartetes Verhalten durch fehlende Synchronisation 155
9.12 Beispielszenario "*Floppy*-Treiber" 156
9.13 Aspekt zum gegenseitigen Aussschluss beim Zugriff auf den DMA Treiber 158
9.14 Erweiterung des Synchronisationsaspekts unter Beachtung des *Floppy*-Treibers 158
9.15 Aspekt zum gegenseitigen Aussschluss beim Zugriff auf den DMA- und den *Floppy*-Treiber 159
9.16 Aspekt zur Leser/Schreiber-Synchronisation entsprechend dem ersten Leser/Schreiber-Problem 160
9.17 Aspekt für den gegenseitigen Ausschluss mit Hilfe eines *Server*-Fadens 162
9.18 Phasen und Objektinteraktion beim Lesen eines Diskettenblocks . 164
9.19 Strategie zur Fadensynchronisation 166

10.1 Schutzkonzepte von AspectJ und AspectC++ im Vergleich 171
10.2 *Firewall* Entwurfsmuster 174
10.3 Das AspectOS Modell 175

Tabellenverzeichnis

2.1 GenVoca Notation für Schichtenbeziehungen 13

4.1 Größe und Kontextwechseldauer verschiedener PURE Konfigurationen 37

5.1 Kompatibilität bei Beziehungen von Aspekten 62

6.1 Umfang der PUMA Subsysteme 80

7.1 Beispiele für Vergleichsmuster 94
7.2 Vordefinierte *Pointcut*-Funktionen 97
7.3 Schnittstelle der Klasse `JoinPoint` für *Advice*-Code 105
7.4 Schnittstelle der Klasse `JoinPoint` bei Einfügungen 107

8.1 Codegröße alternativer Klassenstrukturen bei einmaliger *Template* Instanziierung 121
8.2 Codegröße alternativer Klassenstrukturen bei zweifacher *Template* Instanziierung 121
8.3 Codegrößen und Laufzeiten verschiedener Speicherverwaltungssystemvarianten 126

9.1 Vergleich des Speicherverbrauchs 153
9.2 Vergleich des Ressourcenverbrauchs der Synchronisationsvarianten 163

10.1 Das “wie” und “wo” globaler Strategien im Betriebssystembau . . 176

Kapitel 1

Einleitung

1.1 Motivation

Wer heutzutage das Wort Betriebssysteme hört, denkt im Allgemeinen zuerst an Windows, Linux oder einige wenige andere Systeme für Arbeitsplatzrechner. Dabei gibt es bedeutende Anwendungsgebiete, für die diese Vielzwecksysteme schlecht oder gar nicht geeignet sind.

Leistungsstarke Parallelrechner, wie sie für extrem aufwendige Berechnungen – etwa bei der Wettervorhersage – benötigt werden, gehören ebenso dazu wie sicherheitskritische Bankrechner oder die in immer mehr Geräten enthaltenen eingebetteten Systeme. Ihnen allen ist gemeinsam, dass sie verglichen mit Arbeitsplatzrechnern ein eingeschränktes Aufgabenfeld haben. Dafür werden an sie aber besondere Ansprüche zum Beispiel bezüglich der Ausnutzung der Rechenleistung oder der Ausfallsicherheit gestellt.

Dies gilt in ganz besonderem Maße für die sogenannten eingebetteten Systeme (engl. *embedded systems*). Das sind Rechnersysteme inklusive dazugehöriger Software, die in ein bestimmtes Produkt integriert sind. Dort haben sie in der Regel Steuerungsaufgaben auszuführen oder müssen für die Interaktion mit einem Benutzer sorgen. Typisch für eingebettete Systeme ist, dass sie nach der Auslieferung des Produkts, in das sie integriert sind, nicht mehr verändert werden können. Dies würde einen enormen logistischen und damit finanziellen Aufwand bedeuten, da sie oft in großer Stückzahl hergestellt werden. Beispiele sind Mikrocontroller-basierte Rechnersysteme in modernen Haushaltsgeräten oder im Automobil.

Während das Betriebssystem für einen typischen Vielzweck-Arbeitsplatzrechner auf eine große Menge potentiell angeschlossener Geräte und ein sehr weites Anwendungsspektrum eingerichtet sein muss, hat es der Entwickler eines eingebetteten Systems leichter: Die Art der zu betreibenden Hardware ist von Anfang an ebenso bekannt wie die zu erfüllende Aufgabe. Dafür hat er oft mit extremer Ressourcenknappheit zu kämpfen. Ganz besonders gilt dies für den Bereich der kleinsten eingebetteten Systeme, die auch tiefst eingebettete Systeme genannt werden, wo vorwiegend 4 und 8 Bit Mikrocontrollersysteme mit

nur wenigen KBytes Hauptspeicher eingesetzt werden. In Anbetracht der Tatsache, dass bereits ein einfaches “hello, world” Programm auf einem PC System mehrere hundert KBytes[1] Speicher einnimmt, wird klar, das es besonderer Techniken bedarf, um für solche Systeme überhaupt Software entwickeln zu können.

Dass der Bereich der kleinsten eingebetteten Systeme große wirtschaftliche Relevanz hat, belegen aktuelle Statistiken. So waren zum Beispiel im Jahr 2000 etwa 80% der hergestellten Mikroprozessoren bzw. Mikrocontroller 4 oder 8 Bit Systeme mit linear steigender Tendenz seit 1990 [107]. Neben dem geringen Preis für solche Bauelemente sind aber auch andere Gründe wie ihre Robustheit und nicht zuletzt das vorhandene Entwickler Know-how für den stetigen Erfolg verantwortlich.

In diesem Bereich ist der Druck, sich mit dem Thema der optimalen Ausnutzung knapper Ressourcen auseinanderzusetzen, besonders groß. Dies äußert sich zum Beispiel in der Liste der verwendeten Programmiersprachen, wo C und Assembler noch heute ganz vorne liegen. Modernere Sprachen wie Java kommen wegen ihres teuren Laufzeitsystems hier nicht in Frage und selbst C++ konnte sich trotz unbestrittener Vorteile nicht gegenüber C durchsetzen.

Vor diesem Hintergrund wird klar, warum im Bereich der kleinsten eingebetteten Systeme kein “Betriebssystem von der Stange” eingesetzt werden kann:

1. Die Anforderungen an Systemdienste, die sich in verschiedenen Projekten zur Entwicklung eines kleinsten eingebetteten Systems ergeben, sind zum Teil sehr unterschiedlich. Herstellern fällt es schwer, hier eine Lösung anzubieten, die einen ausreichend großen Markt abdeckt.

2. Klassische Betriebssysteme bieten deutlich mehr Dienste als in diesem Bereich benötigt werden. Dies zu ignorieren und sie trotzdem einzusetzen, wäre mit einer hier inakzeptablen Verschwendung von Ressourcen in Form von Speicherplatz, Rechenleistung und Energieverbrauch verbunden.

Hinzu kommt, dass an der Entwicklung eingebetteter Systeme in vielen Fällen Personen arbeiten, deren Verständnis des Begriffs “Betriebssystem” von den Aufgaben geprägt ist, die ein Betriebssystem im Arbeitsplatzrechnerbereich gewöhnlich wahrnimmt. Dies wird an dem folgenden Zitat deutlich:

> “An operating system (os) is a program that provides an environment for executing other programs, often providing facilities for i/o, a filesystem, networking, virtual memory, and multitasking in a way to run multiple programs concurrently on a single processor.
> Simple embedded systems do not need an operating system. They run a single program that communicates directly with their peripherals. . . ”
>
> S. A. Edwards [38]

[1] 352456 Bytes mit gcc 2.96 unter RedHat Linux 7.1 für x86

Aus diesen Gründen entscheiden sich viele Ingenieure und Programmierer im Bereich der kleinsten eingebetteten Systeme für eine Eigenentwicklung, obwohl bestimmte elementare Betriebssystemdienste wie Unterbrechungssynchronisation, Prozessorzuteilung oder Interprozesskommunikation immer wieder benötigt werden. Damit wird das Rad zwangsläufig jedes Mal neu erfunden.

1.2 Zielsetzung der Arbeit

Um auch beim Bau von Spezialzweckbetriebssystemen ein hohes Maß an Wiederverwendbarkeit zu erreichen, hat es sich die Arbeitsgruppe Betriebssysteme und Verteilte Systeme der Otto-von-Guericke-Universität Magdeburg zum Ziel gesetzt, Methoden und Werkzeuge für diesen Bereich zu entwickeln, wobei ein besonderer Schwerpunkt auf eingebetteten Systemen liegt.

Bemerkenswerte Erfolge konnten bereits durch die Strukturierung der Betriebssystemfunktionen als Programmfamilie und deren objektorientierte Umsetzung mittels C++ erreicht werden. Die nach diesen Konzepten beispielhaft entwickelte Betriebssystemfamilie PURE erschließt durch die extrem feingranulare Abstufung ihrer als Bibliothek angebotenen Funktionen sehr weitreichende Konfigurationsmöglichkeiten. Um dabei den Überblick behalten zu können, erfolgt die Beschreibung der möglichen Systemeigenschaften und deren Auswahl auf Basis sogenannter Feature Modelle. Damit ist es möglich, Beziehungen zwischen Eigenschaften herzustellen und nur sinnvolle Kombinationen zuzulassen. Die anwendungsspezifische Anpassung führt bis hin zu "maßgeschneiderten" Betriebssystemen, die bezüglich Effizienz und Ressourcenverbrauch durchaus mit Speziallösungen konkurrieren können.

Es gibt aber auch Eigenschaften, die sich schlecht in die Struktur einer als funktionalen Hierarchie aufgebauten Programmfamilie einordnen lassen. Das Ziel der vorliegenden Arbeit bestand daher darin zu analysieren, aus welchen Gründen bestimmte Systemeigenschaften bisher nicht konfigurierbar modelliert werden konnten und inwiefern das Konzept der aspektorientierten Programmierung hier Abhilfe schaffen könnte. In der Konsequenz sollten die erforderlichen Methoden und Werkzeuge entwickelt werden, die die praktische Umsetzung der neuen Ideen ermöglichen. Als Testumgebung diente dabei wiederum die Betriebssystemfamilie PURE.

1.3 Gliederung

Es folgt nun ein kurzer Ausblick auf die Kapitelinhalte:

Kapitel 1: "Einleitung" (S. 1–5)
Das Einleitungskapitel steckt grob das Forschungsgebiet der Arbeit ab. Es erläutert die zugrunde liegende Motivation und das Umfeld, das die Konzepte geprägt hat. Die Ausführungen zum Inhalt der Arbeit schließen das

Kapitel ab und geben bereits vorab einen ersten Überblick über die geleisteten Arbeiten und welche Themen ausgeklammert werden mussten.

Kapitel 2: “Grundbegriffe” (S. 7–20)
Das zweite Kapitel erläutert die für die vorliegende Arbeit wichtigen Grundbegriffe objektorientierte Programmierung, Programmfamilien und aspektorientierte Programmierung.

Kapitel 3: “Stand der Kunst” (S. 21–31)
Dieses Kapitel stellt den Stand der Kunst in den Forschungsbereichen, die das Themengebiet dieser Arbeit betreffen, dar. Dabei geht es vor allem um Arbeiten, bei denen das Familienkonzept oder Aspektorientierung auf Betriebssysteme angewendet wird. Darüber hinaus wird untersucht, welche Programmiersprachen die aspektorientierte Programmierung unterstützen und gleichzeitig im Kontext von Betriebssystemen und kleinsten eingebetteten Systemen eingesetzt werden können.

Kapitel 4: “Problemanalyse und Lösungsansatz” (S. 33–49)
Im vierten Kapitel wird erörtert, wo die Probleme bei der Entwicklung einer Betriebssystemfamilie liegen, die durch minimalen Ressourcenverbrauch auch im Bereich der kleinsten eingebetten Systeme eingesetzt werden kann. Unter Berücksichtung der in den vorangegangenen Kapiteln präsentierten Ansätze wird ein grobes Lösungskonzept entwickelt, das allen weiteren präsentierten Ideen und Werkzeugen zugrunde liegt. Dieses Konzept beruht auf der Anwendung der aspektorientierten Programmierung bei der Erstellung von Betriebssystemfamilien.

Kapitel 5: “AOP in Programmfamilien” (S. 51–66)
In diesem Kapitel wird das Lösungskonzept aus Kapitel 4 verfeinert, indem die notwendigen Modelle und Werkzeuge für den aspektorientierten Entwurf von Programmfamilien und die implementierungstechnische Umsetzung erörtert werden.

Kapitel 6: “PUMA” (S. 67–84)
Gegenstand des sechsten Kapitels ist ein im Rahmen der Arbeit entstandenes Analyse- und Transformationssystem für C++ Code namens PUMA. Dieses System stellt die technische Grundlage für alle zur Verfolgung des Ziels nötigen Werkzeuge dar, die in den Kapiteln 7 und 8 vorgestellt werden.

Kapitel 7: “AspectC++” (S. 85–118)
Thema dieses Kapitels ist AspectC++, eine im Rahmen dieser Arbeit entstandene aspektorientierte Spracherweiterung zu C++. AspectC++ stellt die derzeit einzige Möglichkeit dar, selbst im Bereich kleinster eingebetteter Systeme aspektorientiert zu programmieren. Damit bildet diese Sprache eine sehr gute Grundlage für konkrete Fallstudien, die in Kapitel 9 beschrieben werden.

Kapitel 8: “Weitere Aspektweber” (S. 119–133)
Das achte Kapitel stellt weitere aspektorientierte Lösungsansätze zur modularen Implementierung bestimmter Systemeigenschaften dar, die nicht von AspectC++ abgedeckt werden können. Auf diese Weise werden die Grenzen der Vielzweckspracherweiterung AspectC++ aufgezeigt. Gleichzeitig lösen die hier vorgestellten Aspektweber SOSP und COMA wichtige Probleme, mit denen Entwickler ressourcensparender Programmfamilien bisher konfrontiert werden.

Kapitel 9: “Fallstudien” (S. 135–168)
Kapitel 9 zeigt anhand mehrerer Beispiele wie die aspektorientierte Programmierung in Betriebssystemen eingesetzt werden kann. Dabei werden mehrere PURE Subsysteme vorgestellt, die mit Hilfe der erarbeiteten Konzepte und Werkzeuge implementiert wurden. Messungen und Betrachtungen der Quelltexte dienen der Bewertung der Konfigurierbarkeit, der Skalierungsfähigkeit und der Wartbarkeit.

Kapitel 10: “Ausblick” (S. 169–179)
Ein Kapitel mit dem Thema “Ausblick” stellt dar, welche neuen Möglichkeiten die Werkzeuge und Konzepte eröffnen, die im Rahmen der Arbeit entstanden sind.

Kapitel 11: “Rückblick und Diskussion” (S. 181–185)
Den Abschluss der Arbeit bildet ein Rückblick auf die geleisteten Arbeiten und die erzielten Ergebnisse sowie eine Diskussion.

Zum Abschluss dieses Kapitels folgen noch einige Hinweise bezüglich der typografischen Konventionen in diesem Text. Programmcode, Namen von Programmfunktionen oder C++ Klassen sind in der Schriftart `Schreibmaschine` gesetzt. Die Schriftart KAPITÄLCHEN wird für Abkürzungen verwendet, die als Wort zu lesen und auszusprechen sind, wie beispielsweise PURE. Wichtige Begriffe werden durch **Fettdruck** hervorgehoben.

Der Autor ist prinzipiell bemüht, in einem deutschen Text auch Fachbegriffe in deutscher Sprache zu verwenden. Für manche Begriffe existieren jedoch keine verständlichen oder allgemein akzeptierten deutschen Übersetzungen, so dass dem mit dem Fachgebiet der Informatik vertrauten Leser die englischen Bezeichnungen bekannter sind. In diesem Fall werden die englischen Begriffe verwendet oder zumindest in Klammern als Übersetzung präsentiert und durch *Schrägschrift* markiert. Klassendiagramme orientieren sich an den Konventionen der Unified Modeling Language (UML [87]).

Kapitel 2

Grundbegriffe

Ein wesentliches Ziel bei den Forschungsaktivitäten der Arbeitsgruppe besteht darin, die Wiederverwendbarkeit von Software auch in den Bereichen zu erhöhen, wo bisher aus Gründen der Ressourcenknappheit fast ausschließlich Speziallösungen geschaffen wurden. Um diese Vorhaben zu realisieren, werden verschiedene Konzepte zum Teil schon länger erfolgreich eingesetzt. Andere Ideen kamen im Rahmen dieser Arbeit hinzu. Die wichtigsten dieser Grundbegriffe sollen im Folgenden vorgestellt werden.

2.1 Objektorientierung

Während in anderen Gebieten wiederverwendbare Software gerne objektorientiert entworfen und implementiert wird, fürchten die Entwickler kleiner eingebetteter Systeme meist, dass damit zusätzliche Kosten an Speicherplatz und Laufzeit entstehen. Im Rahmen der Betriebssystemfamilie PURE konnte jedoch gezeigt werden, dass bei umsichtiger Verwendung der teuren Konstrukte auch objektorientierte Implementierungen möglich und für die Entwicklung konfigurierbarer Software besonders geeignet sind.

In objektorientierten Programmiersprachen werden Daten und die auf diesen Daten arbeitenden Operationen gemeinsam in Form von Objekten gekapselt. Gleichartige Objekte, das heißt solche mit gleicher Datenstruktur und gleichen Operationen, werden durch Klassen beschrieben.

2.1.1 Vererbung

Die Vererbung ist ein Konzept, das es erlaubt, eine neue Klasse zu erstellen, die die Eigenschaften – also Datenstruktur und Operationen – einer bereits existierenden Klasse erbt. Man spricht in diesem Zusammenhang auch von der sogenannten Basisklasse und der erbenden abgeleiteten Klasse. Abbildung 2.1 auf der nächsten Seite zeigt, wie diese Beziehung in einem Klassendiagramm dargestellt werden kann.

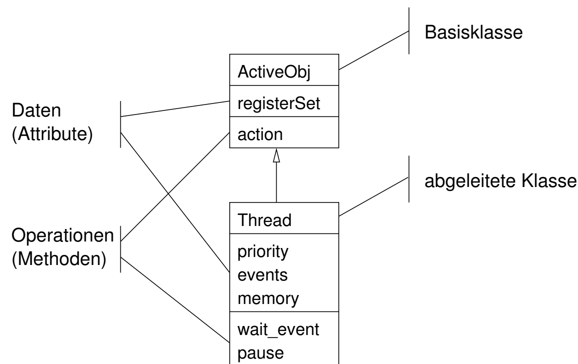


Abbildung 2.1: Vererbungsbeziehung im Klassendiagramm

Durch die Vererbung kann eine neue Klasse unter Wiederverwendung einer existierenden Basisklasse erstellt werden. Alle öffentlichen Operationen der Basisklasse sind auch auf Objekten der abgeleiteten Klasse anwendbar. Ebenso ist die komplette Datenstruktur der Basisklasse automatisch in der abgeleiteten Klasse enthalten. Die abgeleitete Klasse kann genutzt werden, um die Eigenschaften der Basisklasse zu erweitern. Da die abgeleitete Klasse alle Eigenschaften der Basisklasse geerbt hat, kann überall dort, wo ein Basisklassenobjekt referenziert wird, auch ein Objekt der abgeleiteten Klassen verwendet werden[1].

Für die Erstellung konfigurierbarer Software ist von besonderer Bedeutung, dass die Verwendung des Vererbungsmechanismus von C++ keine zusätzlichen Kosten verursacht. Ein Objekt, wie beispielsweise der für die Prozessverwaltung typischerweise vorhandene Kontrollblock, kann also gleichermaßen durch eine einzelne Klasse wie durch eine mehrstufige Vererbungshierarchie ausgedrückt werden. Die Modellierung der Daten als Objekte einer Klasse, die in mehreren Schritten von einer Basisklasse abgeleitet wurde, erlaubt dabei nicht nur eine saubere Strukturierung, sie vereinfacht auch die Konfigurierung. Statt nämlich stets Objekte der am weitesten spezialisierten Klasse zu instanziieren, können bei geringeren Anforderungen auch Objekte aus der Mitte der Vererbungshierarchie verwendet werden. Der Umfang des Systems an Code und Daten wird dann entsprechend geringer. Dabei ist es nicht einmal notwendig, im Quelltext Konfigurierungspunkte, etwa zur bedingten Übersetzung, vorzusehen, da in diesem Fall die Konfigurierung automatisch durch den Binder erfolgt. Voraussetzung ist allerdings, dass bei der Bestimmung der Reihenfolge, in der die verschiedenen Eigenschaften hinzugefügt werden, genügend Weitsicht bewiesen wurde.

[1] Diese Eigenschaft wird als *Liskov Substitution Principle* bezeichnet [74].

2.1.2 Dynamisches Binden

Objektorientierte Programmiersprachen erlauben es, dass Operationen aus Basisklassen in abgeleiteten Klassen überdefiniert werden. Wenn nun ein Objekt über eine Referenz eine Operation eines Basisklassenobjekts aufruft, tatsächlich aber die Referenz auf eine Instanz einer abgeleiteten Klasse verweist, wird die in der abgeleiteten Klasse definierte spezialisierte Variante der Operation ausgeführt. Da von einer Klasse mehrere Klassen abgeleitet sein können und die Entscheidung, welche Operation auszuführen ist, zur Laufzeit anhand des Objekttyps, das heißt anhand seiner Klasse, erfolgt, wird hier von dynamischem Binden gesprochen.

In besonderem Maße wird das dynamischen Binden nützlich, wenn mit Schnittstellenklassen (engl. *interfaces*) gearbeitet wird. Solche Klassen definieren keine Implementierungen für die deklarierten Operationen. Stattdessen geben sie nur eine Schnittstelle vor. Sie können nicht selbst für die Objektinstanziierung genutzt werden. Die Implementierung erfolgt erst in den abgeleiteten Klassen. Dafür können auch Objekte unterschiedlicher Klassen, die dieselbe Schnittstelle realisieren, in vielerlei Hinsicht gleich behandelt werden, zum Beispiel könnten in eine Listenklasse, die über die Schnittstellenklasse `ListElement` auf ihre Elemente zugreift, alle Objekte von Klassen eingeordnet werden, die von `ListElement` Erben und die deklarierten Methoden der Schnittstelle implementieren.

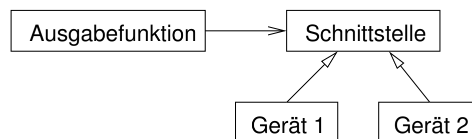


Abbildung 2.2: Struktur des Strategiemusters

Das dynamische Binden vereinfacht auch die Konfigurierung des Systems. So können beispielsweise Ausgabefunktionen, die hardwareunabhängig sein sollen, eine abstrakte Treiberschnittstelle nutzen, wie es in Abbildung 2.2 gezeigt ist. Die tatsächliche Verbindung zu einem konkreten Gerätetreiber kann so zum Zeitpunkt der Systemkonfigurierung oder auch erst zur Laufzeit erfolgen. Diese Flexibilität wird allerdings mit zusätzlichen Kosten an Speicherplatz und Laufzeit erkauft.

2.1.3 Entwurfsmuster

Besonders ausgeprägt werden Schnittstellenklassen in den sogenannten Entwurfmustern [46] genutzt. Diese werden im Zusammenhang mit objektorientierter Software verwendet, um eine Art Musterlösung für immer wiederkehrende Entwurfsprobleme vorzugeben. Auf diese Weise kommen weniger erfahrene Softwareentwickler in den Genuss erprobter Softwarestrukturen und Entwicklungsteams bekommen ein Vokabular, das die Kommunikation über geeignete objektorientierte Systemstrukturen erleichtert.

Entwurfsmuster bestehen hautpsächlich aus einem Klassendiagramm, bei dem die Klassennamen als Platzhalter zu verstehen sind. Dazu gehört außerdem noch eine ausführliche Erläuterung, wann dieses Muster sinnvoll einzusetzen ist und unter welchen Umständen sich ein Einsatz nicht empfiehlt.

Das typische Entwurfsmuster folgt dabei dem Grundgedanken, dass die Softwarestrukturen einer Erweiterung oder Veränderung des Systems nicht im Wege stehen dürfen. Daher werden sehr flexible Objektbeziehungen vorgeschlagen, die sich auf der Implementierungsseite nur durch starken Einsatz dynamischen Bindens realisieren lassen.

Subsysteme eines anwendungs- oder plattformspezifisch anpassbaren Betriebssystems können durch solche Techniken sehr lose gekoppelt und damit unabhängig von einander konfiguriert werden. Allerdings müssen hierbei in ganz besonderem Maße die durch das dynamische Binden verursachten Kosten im Auge behalten werden.

2.2 Programmfamilien

Das Konzept der Programmfamilie entstand bereits Mitte der 70er Jahre. Dabei bezogen sich schon die ersten Beispiele auf das Gebiet der Betriebssysteme. So stammten wichtige Ideen, die in Richtung von Programmfamilien gehen, aus dem T.H.E. System [35]. Später wurde dann mit FAMOS[2] [50] eine erste Betriebssystemfamilie vorgestellt. Die damals angestellten Überlegungen gelten nach wie vor und auch die geprägten Begriffe [89] finden noch heute Verwendung.

2.2.1 Grundlagen

Der wesentliche Grundbegriff des familienbasierten Entwurfs ist die **Funktion** des zu realisierenden Systems. Eine Funktion ist in diesem Kontext eine Leistung oder ein Dienst, der von einem Softwaresystem erbracht wird. Üblicherweise können Funktionen in Teilfunktionen zerlegt werden (**funktionale Dekomposition**). Dann ergibt sich die Gesamtfunktion aus der Summe der Teilfunktionen. Bei einer Programmfamilie ist es schwierig, von einer Gesamtfunktion zu sprechen, da es sehr unterschiedliche Familienmitglieder geben kann. Die Familienmitglieder unterscheiden sich durch die Teilfunktionen aus denen sie gebildet werden und damit in ihrer Gesamtfunktion.

Der Vorteil bei der Zerlegung einer Funktion in Teilfunktionen, die sich gegenseitig benutzen, besteht in der Möglichkeit der **Wiederverwendung**. Bei Programmfamilien spielt dies eine besonders große Rolle, da neue Familienmitglieder mit möglichst wenig Aufwand realisiert werden sollen, indem Funktionen von existierenden Familienmitgliedern wiederverwendet werden. Dies setzt aber eine entsprechende Strukturierung der Software voraus.

[2] FAMOS steht für *Family of Operating Systems*

Ein hohes Maß an Wiederverwendung kann erreicht werden, wenn das System so entworfen wurde, dass Funktionen sich gegenseitig benutzen. Das gegenseitige Benutzen kann aber auch zum Nachteil werden, wenn die Beziehungen zu komplex werden, so dass keine unabhängigen Teilfunktionen mehr existieren. In diesem Fall kann das System erst getestet werden, wenn alle Teilfunktionen implementiert sind.

Das extreme Gegenteil wäre ein System bei dem die Teilfunktionen unabhängig voneinander sind. Dafür sind dann aber innerhalb der Teilfunktionen bestimmte andere Funktionen, die an sich wiederverwendbar wären, redundant enthalten.

Der Kompromiss beider Varianten besteht darin, die gegenseitige Benutzung von Funktionen so zu beschränken, so dass keine Zyklen in den Abhängigkeitsbeziehungen entstehen. Die so entstehenden Graphen werden als **Funktionale Hierarchie** bezeichnet. Als Beispiel zeigt Abbildung 2.3 die funktionale Hierachie des bereits angesprochenen FAMOS Systems.

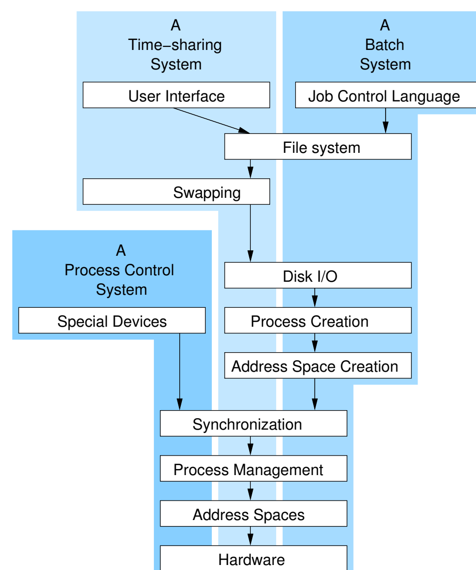


Abbildung 2.3: Die Funktionale Hierarchie des FAMOS Systems

FAMOS war ein aus Schichten aufgebautes System sich schrittweise erweiternder virtueller Maschinen. Jede der Schichten kennt die darunterliegenden Schichten und benötigt sie für die Erbringung der eigenen Aufgabe. Die beschrifteten Rechtecke in der Abbildung repräsentieren die Funktionen des Systems. Durch einfache Konfigurierungsmaßnahmen konnten daraus drei verschiedene Familienmitglieder erzeugt werden: Ein *Process Control* System, ein *Time-sharing* System und ein *Batch* System. Durch die Abhängigkeitsbeziehungen in der funktionalen Hierarchie konnte beispielsweise modelliert werden, dass für die *Special Devices* in der *Process Control System* Variante von FAMOS auch die Funktionen *Synchronization*, *Process Management*, *Address Spaces* und *Hardware* gebraucht werden. Diese vier Basisfunktionen werden ebenfalls für die anderen beiden Familienmitglieder

benötigt. Dafür kann dort auf die Funktion *Special Devices* verzichtet werden.

Auf den ersten Blick mag die Zerlegung einer Software in Schichten, wie sie hier vorgenommen wurde, trivial wirken. Tatsächlich erfordert dies jedoch ein großes Maß an Erfahrung. Zum Beispiel wurden bei FAMOS die Funktionen Prozessverwaltung (*Process Management*) und Prozesserzeugung (*Process Creation*) voneinander getrennt, obwohl beide letztlich mit den gleichen Datenstrukturen arbeiten. Dadurch wird erreicht, dass in der *Process Control* Systemkonfiguration kein Ballast in Form ungenutzten Programmcodes entsteht, denn eine dynamische Prozesserzeugung wird hier nicht gebraucht. Beim Entwurf jeder einzelnen Funktion bzw. Schicht muss also versucht werden, Entwurfsentscheidungen zu vermeiden, die der Erweiterung der Familie im Wege stehen könnten. Dies verlangt viel Weitblick und Disziplin.

Zudem sollte der Entwurf, um eine möglichst genaue Anpassung an das jeweilige Einsatzszenario zu gestatten, von einer **minimalen Basis** ausgehen, die mit **minimalen Erweiterungen** schrittweise funktional angereichert wird. Auch das Suchen der minimalen funktionalen Einheiten macht den familienbasierten Entwurf schwer.

2.2.2 GenVoca

Seit den 70er Jahren hat sich das Konzept der Programmfamilie weiterentwickelt. Es wird vor allem verwendet, um "Wiederverwendung im Großen" zu erreichen. Auf diese Weise ergänzt es elegant die Konzepte der Objektorientierung, die lediglich Wiederverwendung auf der Ebene von Klassen ermöglichen.

Objektorientierte virtuelle Maschinen

Das GenVoca Modell [10] baut heute auf dem Konzept der objektorientierten virtuellen Maschinen (OOVM) auf. Abbildung 2.4 auf der nächsten Seite soll andeuten, wie nach diesem Modell die Funktionen der funktionalen Hierarchie durch eine Menge kollaborierender Klassen realisiert werden können. Jede Ebene bildet eine objektorientierte virtuelle Maschine. Die Klassen der Ebene i dürfen nur Klassen der Ebenen 0 bis i benutzen.

Modellierung von Familien mit GenVoca

Mit GenVoca ist es nun möglich, die Beziehungen zwischen OOVMs zu beschreiben. Es wird davon ausgegangen, dass jede Schicht in einem solchen System für die darüberliegenden Schichten eine Schnittstelle bereitstellt, über die ihre Funktionen zugänglich sind. Ein solche Schnittstelle besteht in der Regel aus mehreren Klassen, die zueinander in Beziehung stehen. Ebenso erwartet eine Schicht, die eine darunterliegende Schicht benutzt, von dieser eine bestimmte Schnittstelle. Tabelle 2.1 zeigt, wie diese Beziehungen beschrieben werden.

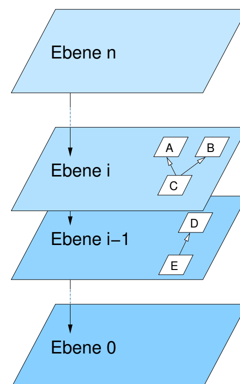


Abbildung 2.4: Eine Hierarchie von OOVMs

Notation	Bedeutung
y: S	Die Schicht 'y' implementiert Schnittstelle 'S'
g[x:S]: R	'g' implementiert 'R' und benutzt eine Schicht, die 'S' implementiert

Tabelle 2.1: GenVoca Notation für Schichtenbeziehungen

Eine Erweiterung von GenVoca gegenüber dem einfachen Modell der funktionalen Hierarchie besteht darin, dass Schichten mit identischen Schnittstellen austauschbar sind. Damit ergeben sich Familienmitglieder nicht nur durch das Weglassen von Ästen oder Schichten, sondern auch dadurch, dass jede Funktion in mehreren schnittstellenkompatiblen Varianten vorliegen kann. Da die Varianten unterschiedlicher Funktionen unabhängig sind, multipliziert sich dies zu einer sehr viel größeren Konfigurationsvielfalt.

Die Menge der kompatiblen Implementierungen einer Schicht wird im GenVoca Vokabular als *Realm* bezeichnet. Als Symbol dafür wird der Name der Schnittstelle benutzt. Zum Beispiel:

```
S = { y, z, w }
R = { g[x:S], h[x:S], i[x:S] }
```

Hiermit wird zum Ausdruck gebracht, dass die Schichtimplementierungen 'y', 'z' und 'w' die Schnittstelle 'S' implementieren und somit ein *Realm* bilden. 'g', 'h' und 's' benutzen die Schnittstelle 'S' und können so mit allen Varianten dieser Schicht verknüpft werden. Den gleichen Sachverhalt könnte man auch durch die folgende Grammatik zum Ausdruck bringen:

```
S := y | z | w
R := g S | h S | i S
```

Jeder Satz, der durch diese Grammatik produziert werden kann, ist ein Familienmitglied. Zum Beispiel:

```
System1 = g[y];
System2 = g[w];
System3 = h[w];
```

Auf diese Weise kann eine Programmfamilie präzise beschrieben werden. Man kann dadurch insbesondere die Eigenschaften einzelner Schichtimplementierungen nutzen, wie zum Beispiel ihren Speicherplatzverbrauch, und damit auf die Eigenschaften der Familienmitglieder schließen, um so zum Beispiel das Mitglied mit dem insgesamt geringsten Speicherplatzverbrauch zu bestimmen[3]. Zu diesem Zweck wird lediglich die Grammatik zu einer Attributgrammatik [1] erweitert. Der Speicherplatzverbrauch kann dort als synthetisiertes Attribut modelliert werden. Daneben kann mit Hilfe von Attributen domänenspezifisches Wissen in das Modell eingebracht werden. So können zum Beispiel bestimmte Systemkonfigurationen auf ihre Sinnhaftigkeit hin überprüft werden [8]. Dies ist immer dann erforderlich, wenn die Schnittstellen der Schichten zusammenpassen, aber andere Bedingungen den gemeinsamen Einsatz in einem System verbieten. Für die Implementierung solcher Tests werden Compilerbauwerkzeuge eingesetzt, die in der Lage sind mit Attributgrammatiken umzugehen.

Implementierung von GenVoca Familien

Für die Implementierung eines Systems auf Basis des GenVoca Modells gibt es viele Varianten. So können die Schichten in Form von Objekten zur Laufzeit instanziiert und verbunden werden oder durch Metaprogramme, Tranformationssysteme oder parametrisierten Code (*Templates*) generiert werden.

Die neueren auf GenVoca basierenden Anwendungsfamilien basieren auf dem Konzept der *Mixin-Layers*. Der Begriff *Mixin* ist in der Informatik stark überladen. Er stammt ursprünglich aus der Programmiersprache CLOS und wird auch im Zusammenhang mit C++ verwendet. Die folgenden Erläuterungen beziehen sich auf das allen zugrunde liegende Konzept [21].

Ein *Mixin* ist eine Klasse, mit der man eine Erweiterung einer anderen Klasse implementiert, ohne dabei zu wissen, wie die zu erweiternde Basis aussieht. Die erweiterte Basisklasse ist ein Parameter bei der *Mixin* Definition. In C++ lässt sich dies leicht durch *Templates* ausdrücken:

```
template <class Base>
class Mixin : public Base
 { /* Rumpf der Mixin Klasse */ };
```

[3]Eine Voraussetzung dafür ist allerdings, dass der Speicherplatzverbrauch einer Schichtimplementierung überhaupt quantifizierbar ist. Dies ist zum Beispiel nicht gegeben, wenn wie bei C++ üblich mit Einbettung von Funktionen (*inlining*) gearbeitet wird. Ebenso lässt sich auch der Bedarf an dynamisch reserviertem Speicher selten vorhersagen.

Dieser Mechanismus kann zur Implementierung von Programmfamilien genutzt werden, wenn er mit dem Konzept der inneren Klassen (engl. *inner classes*) verknüpft wird [99]. So kann eine *Mixin* Klasse eine ganze Schicht aus einer Hierarchie von OOVMs implementieren und man spricht von einer *Mixin* Schicht (engl. *mixin layer*):

```
template <class BaseLayer>
class ThisLayer : public BaseLayer
 {
   public:
      class Mixin1 : public BaseLayer::Mixin1
       { /* ... */ };
      class Mixin2 : public BaseLayer::Mixin2
       { /* ... */ };
 };
```

Solche *Mixin* Schichten können über die *Template* Parametrisierung zur Übersetzungszeit miteinander verbunden werden. Da von einer benutzten Schicht lediglich die Schnittstelle, aber nicht der Name bekannt sein muss, können Schichtimplementierungen entsprechend der Regeln der GenVoca Grammtik ausgetauscht werden. Die Selektion eines Familienmitglieds kann einfach mit Hilfe einer Typdefinition erfolgen:

```
typedef Layer_R_g<Layer_S_y> System1; // oder
typedef Layer_R_g<Layer_S_w> System2; // oder
typedef Layer_R_h<Layer_S_w> System3;
```

2.2.3 Feature-basierte Konfigurierung

Weitere wichtige Impulse für die familienbasierte Softwareentwicklung kamen vor wenigen Jahren durch die Verbindung mit dem aus der Feature-orientierten Domänenanalyse (FODA) [62] stammenden Konzept der Feature Modelle [32]. Mit Feature Modellen können die Gemeinsamkeiten und Unterschiede der Eigenschaften der Mitglieder einer Programmfamilie beschrieben werden. Eigenschaften können als “optional” oder “notwendig” markiert und durch Relationen wie “alternativ” und “oder” zueinander in Beziehung gesetzt werden.

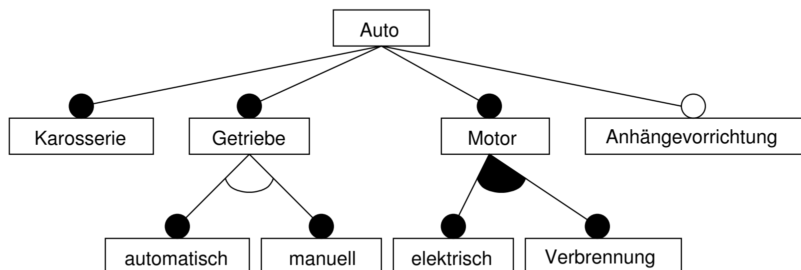


Abbildung 2.5: Ein Feature Diagramm des Konzepts “Auto”

Abbildung 2.5 auf der vorherigen Seite zeigt als Beispiel ein Feature Diagramm des Konzepts “Auto”. Der gefüllte Bogen zwischen den Eigenschaften “elektrisch” und “Verbrennung” zeigt eine “oder” Beziehung an. Der Motor eines Autos kann also elektrisch sein, auf Verbrennung beruhen oder im Fall eines hybriden Antriebs auch beides. Der nicht gefüllte Bogen zwischen “automatisch” und “manuell” repräsentiert eine “alternativ” Beziehung. Eigenschaften mit einem gefüllten Punkt sind notwendig, während solche mit nicht gefülltem Punkt – wie Anhängevorrichtung – optional sind. Neben dem Feature Diagramm gehören zu einem Feature Modell noch weitere Informationen wie Feature Beschreibungen oder weitere Einschränkungen bezüglich der Feature Auswahl [30].

Die Abbildung der abstrakten Features aus der Problemdomäne in konkrete Implementierunskomponenten erfolgt mit Hilfe von zusätzlichem Konfigurationswissen. Es konnte gezeigt werden [32], dass dieses Wissen beispielsweise durch eine C++ Generatorklasse ausgedrückt werden kann, die mit Hilfe der *Template*-Metaprogrammierung [111][108] zur Übersetzungszeit anhand von Features und eingebauten Abbildungsregeln die richtigen Schichten einer GenVoca Architektur auswählt. Ob allerdings *Template*-Metaprogramme für diese Aufgabe die beste Wahl sind, darf bezweifelt werden. So kommt es zu einer Durchmischung von Konfigurierungs- und Komponentencode, was die Übersichtlichkeit beeinträchtigt. Darüber hinaus ist die Erstellung einer solchen Generatorklasse für größere Systeme eine sehr schwierige Aufgabe, die für einen durchschnittlichen Programmierer durchaus zu einer unüberwindlichen Hürde werden kann.

Feature Modelle ergänzen jedoch auf der Analyseseite das Modell der funktionalen Hierarchie sehr gut, das eher in der Entwurfsphase eingesetzt wird. Es eignet sich außerdem auch gut für die Konfigurierung einer Familie, da ein Anwendungsentwickler, der ein Betriebssystem konfiguriert, ein gutes Verständnis von dem Vokabular in einem Feature Modell hat, da sich dieses an Systemeigenschaften und nicht an Systemmodulen orientiert.

2.3 Aspektorientierte Programmierung

Die aspektorientierte Programmierung (AOP) ist eine von mehreren neuen Programmiertechniken, die zum Ziel haben, die Ausdruckskraft der Programme, verglichen mit herkömmlichem, beispielsweise rein objektorientiertem oder prozeduralem Code, zu steigern. Beispiele für solche Ansätze sind die generative und generische Programmierung, domänenspezifische Sprachen, Reflektion und Metaprogrammierung[4].

Aspektorientierte Programmiersprachen oder -werkzeuge erreichen diese gesteigerte Ausdruckskraft durch die Eigenschaft, dass ein einzelner separater Teil eines Programms Auswirkungen auf viele andere Teile des Programmsystems haben kann, und dass die Programmteile, auf die ein solcher separater Programmteil

[4]Eine gute Übersicht über all diese Ansätze liefern K. Czarnecki und U. Eisenecker in ihrem Buch [33].

einwirkt, nicht speziell dafür präpariert werden müssen. Der Entwickler muss also keine Ankerpunkte vorsehen. Man kann dieses Verhalten auch als implizite Ausführung bezeichnen [39]. So ist es zum Beispiel möglich zu verlangen, dass eine festgelegte Aktion ausgelöst wird, immer bevor eine bestimmte Funktion ausgeführt wird, ohne dass dies dem Programmcode des Aufrufers oder der aufgerufenen Funktion anzusehen ist.

Durch diese Eigenschaft wird es möglich, die sogenannten *Crosscutting Concerns* in modularer Weise zu implementieren. Dies sind Entwurfsentscheidungen, die zu Programmcode führen, der nicht in das Modularisierungsschema des restlichen Programms passt, und so über weite Teile des Programms verstreut wird. Statt einen logisch zusammenhängenden Programmcode in Fragmente zu zerlegen und an verschiedenen Stellen eines Programms einzusetzen, wird die implizite Ausführung genutzt. Die Module, die dies leisten, werden **Aspekte** genannt und bestehen aus den auszuführenden Codefragmenten selbst und einer Beschreibung, von welchen Punkten aus die Ausführung stattfinden soll. Ein solcher Punkt wird als Verbindungspunkt (engl. *join point*) bezeichnet.

Aspekte können zum Beispiel benutzt werden, um quasi von außen Code zur Überprüfung von Vor- und Nachbedingungen von Funktionen in ein System einzubringen. Häufig sind solche Bedingungen für eine große Zahl von Funktionen identisch, so dass eine modulare, aspektorientierte Implementierung weniger Programmtext bedeutet. Gleichzeitig erhöht sich die Übersichtlichkeit der einzelnen Funktionen und am Ende der Entwicklungsphase kann der Aspekt sehr leicht entfernt werden. Aspekte können aber auch sinnvoll im produktiven Betrieb eingesetzt werden. Ein häufig genanntes Beispiel sind hier Aspekte, die für die Synchronisation innerhalb einer Komponente sorgen, wenn diese in einer vielfädigen Umgebung eingesetzt wird.

Den technischen Vorgang, der für die implizite Ausführung des Aspektcodes an den Verbindungspunkten sorgt, nennt man **Aspektweben**. Der **Aspektweber** kann zum Beispiel ein Codetransformationssystem sein, das Funktionsaufrufe generiert, ein Compiler, der entsprechende Aufrufe als Maschineninstruktionen einfügt, oder ein Laufzeitsystem, das in einen Interpreter oder eine virtuelle Maschine eingebunden ist und dort Verbindungspunkte zur Laufzeit erkennt und mit Aspektcode verbindet. Abbildung 2.6 auf der nächsten Seite illustriert den Vorgang des Aspektwebens. Dort liegen der Komponentencode, der aus der üblichen funktionalen Zerlegung des Gesamtproblems resultiert, und der Aspektcode getrennt im Sinne des Prinzips *Separation of Concerns* vor. Erst durch den Aspektweber werde beide Teile verbunden.

Abbildung 2.7 auf der nächsten Seite zeigt links die Implementierung eines *Crosscutting Concerns* ohne und rechts mit Hilfe von Aspekten. Die Rechtecke symbolisieren die verschiedenen Quelltexteinheiten, die zu dem Programmsystem gehören. Der Code, der für die Implementierung des *Crosscutting Concerns* benötigt wird, ist hervorgehoben. Wie zu erkennen ist, führt die Lösung mit Aspekten auf der rechten Seite zu einer verbesserten Modularität.

Die Anwendung von Aspekten soll die Lesbarkeit des Codes und damit seine

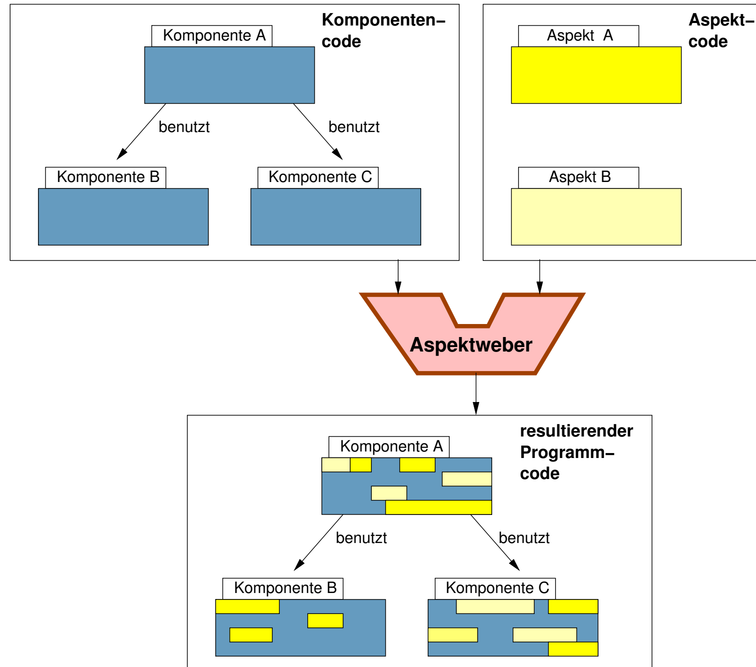


Abbildung 2.6: Der Vorgang des Aspektwebens

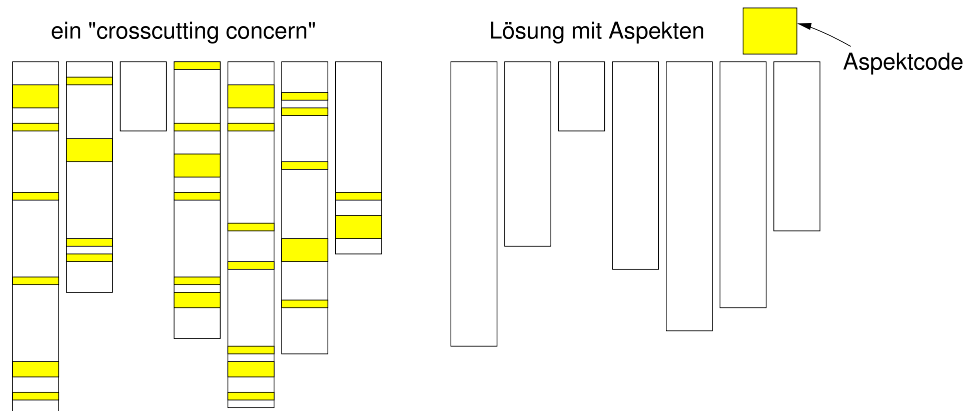


Abbildung 2.7: Modulare Implementierung eines *Crosscutting Concerns*

Wartbarkeit erhöhen. Zudem soll die zu implementierende Codemenge reduziert und damit die Produktivität der Entwickler gesteigert werden. Erste Studien legen nahe, dass solche Verbesserungen tatsächlich mit AOP zu erreichen sind [83].

2.3.1 AspectJ

AspectJ[5] [66] ist neben Hyper/J[6] [88] die am weitesten fortgeschrittene und populärste Programmiersprache, die aspektorientierte Programmierung unterstützt. Es handelt sich bei beiden Sprachen um Erweiterungen zu Java.

Die wichtigsten Erweiterungen sind die Sprachelemente Aspekt, *Pointcut* und *Advice*. Aspekte sind eine Erweiterung des Konzepts der Klassen und dienen als modulare Einheit zur Implementierung eines *Crosscutting Concerns*. Ein *Pointcut* repräsentiert eine Menge von Verbindungspunkten und ist damit das wichtigste Sprachelement, um ein *Crosscutting Concern* zwecks Modularisierung in den Griff zu bekommen. *Advice* ist eine Aktion, die ausgeführt werden soll, wenn ein Verbindungspunkt eines bestimmten *Pointcuts* erreicht wurde.

Ein Verbindungspunkt in AspectJ ist nicht statisch als Punkt im Quellcode, sondern als Punkt im Kontrollfluss des laufenden Programms definiert. Er besitzt also neben der Orts- noch eine Zeitkoordinate. So kommt es, dass bei gewissen Arten von *Pointcuts* erst zur Laufzeit festgestellt wird, ob an einem bestimmten Punkt im Code ein *Advice* aktiviert werden muss. Ein solcher dynamischer Verbindungspunkt ist zum Beispiel der Aufruf einer bestimmten virtuellen Methode auf einem Objekt einer bestimmten Klasse. Das heißt, die Aktivierung des *Advice* hängt vom Zielobjekttyp des Aufrufs ab. Vor dem Hintergrund des dynamischen Bindens bleibt in diesem Fall eine Typüberprüfung zur Laufzeit nicht aus.

Neben den dynamisch zu sehenden *Pointcuts*, die mit *Advice* verbunden werden, unterstützt AspectJ auch rein statische Systemerweiterungen. Diese sogenannten *Introductions* fügen zum Beispiel eine neue Methode oder ein neues Attribut in eine beliebige Menge von Klassen ein, die durch eine Art regulären Ausdruck für Typen beschrieben werden, der GTN (*generalized type name*) genannt wird.

Das folgende Codefragment stammt aus einem AspectJ *Tutorial* [65] und zeigt einen einfachen Aspekt zur Kontrollflussverfolgung. Spezielle AspectJ Schlüsselworte oder vordefinierte Namen sind hervorgehoben.

```
aspect SimpleTracing {
  pointcut traced():
    call(void Display.update()) ||
    call(void Display.repaint(..));

  before(): traced() {
    system.out.println("Entering" + thisJoinPoint);
  }
}
```

[5] http://aspectj.org/

[6] http://www.research.ibm.com/hyperspace/

Verfolgt werden Aufrufe der Methoden `update()` und `repaint()` der Klasse `Display`. Beschrieben wird dies durch die Defintion des *Pointcuts* `traced()`. Das mit `before()` eingeleitete Codefragment ist ein *Advice* und soll ausgeführt werden, wenn der Kontrollfluss einen der Verbindungspunkte aus `traced()` erreicht, das heißt, wenn `Display.update()` oder `Display.repaint()` mit beliebiger Argumentliste aufgerufen wird. Wie bereits erwähnt, geschieht dies, ohne dass der Aufrufer oder die aufgerufene Klasse dafür prapariert werden müssen. Neben dem Erweitern ohne Eingriffe in existierenden Code erlaubt diese Implementierung in Verbindung mit statischen Konfigurierungsmaßnahmen das einfache An- und Abschalten der Kontrollflussverfolgung.

An diesem Beispiel fällt bereits eine kritische Eigenschaft von AspectJ Programmen auf, die mit dem *Pointcut*-Konzept zusammenhängt und dadurch auch in anderen aspektorientierten Sprachen auftreten kann. Durch die Nennung der Namen `Display`, `update` und `repaint` gibt es innerhalb der Klasse `Display` eine unsichtbare Abhängigkeit. Ein Programmierer könnte durch eine einfache Umbennung dafür sorgen, dass die Observierung durch den Aspekt nicht mehr funktioniert. Aus diesem Grund sind die AspectJ Entwickler bemüht, integrierte Java Entwicklungsumgebungen so zu erweitern, dass Verbindungspunkte im Quellcode markiert werden und damit die Beziehung zum Aspekt sichtbar wird.

Kapitel 3

Stand der Kunst

Dieses Kapitel dient dazu, den aktuellen Stand der Forschung und Forschungsaktivitäten in den Themenbereichen Programmfamilien und aspektorientierte Programmierung im Umfeld von Systemsoftware darzustellen. Ein Fazit am Ende des Kapitels fasst alles zusammen und beschreibt so die Ausgangssituation für die in den folgenden Kapiteln beschriebenen eigenen Arbeiten.

3.1 Produktlinien

Der Begriff Programmfamilie, wie er in Kapitel 2 verwendet wurde, wird heute nur noch selten benutzt. Stattdessen wird von **Produktlinien** gesprochen, was Programmfamilien einschließt. Dieses Forschungsgebiet ist hochaktuell und wird auf zahlreichen Softwaretechnik Konferenzen – vor allem der *Software Product Line Conference* (SPLC) Serie – diskutiert.

Die Forschungsthemen sind in diesem Bereich vielfältig. Es geht hier um die Variabilität von Produktliniensoftware im Allgemeinen, Softwarearchitekturen, ihre Implementierungskomponenten und auch um rein organisatorische Fragen im Unternehmen, die sich aus der Anwendung von Produktlinien ergeben. Auch das bereits angesprochene Thema *Feature Modelling* und Methoden zur Domänenanalyse wie FODA [62] oder sein Nachfolger FORM [63] spielen eine wichtige Rolle.

Einige (wenige) Arbeiten in diesem Bereich zeigen die Sinnhaftigkeit der Verbindung von aspektorientierter Programmierung und Produktlinien auf [48][32]. Wesentliches Argument dabei ist, dass die Evolution einer Produktlinie erheblich erschwert wird, wenn eine einzelne Feature-Auswahl zu Konfigurierungseingriffen in vielen verschiedenen Implementierungskomponenten führt. Daneben wurde auch die Idee aufgebracht, bereits auf der Ebene der Domänenanalyse Aspekte explizit zu behandeln [68].

Eine Verbindung zwischen Produktlinien und Betriebssystemen stellen Arbeiten aus dem GeneSys Projekt[1] (Universität Kaiserslautern) her [11]. Darin wird vorgeschlagen, Systeme aus generischen Komponenten aufzubauen, die anpassbar

[1] http://wwwagss.informatik.uni-kl.de/

sind und deren Variabilität durch sogenannte *Design Spaces* [70][12] beschrieben wird. Für die Anpassung selbst wurden spezielle Präprozessoren und Codegeneratoren verwendet. AOP kam nicht zum Einsatz.

3.2 Programmspezialisierung

Ein weiterer Ansatz, Anpassungen von Software gerade auch im Betriebssystembereich vorzunehmen, ist die Programmspezialisierung. Darunter wird im Allgemeinen das Erzeugen einer spezieller Programmcodevariante aus einem generischen, für viele Anwendungsszenarien ausgelegten Programmcode verstanden. Dazu muss von außen eine Menge von sogenannten Spezialisierungsprädikaten angegeben werden. Sie beschreiben die im jeweiligen Szenario geltenden Einschränkungen gegenüber dem allgemeinen Anwendungsfall. Unter Beachtung der Prädikate wird dann der generische Code zum Spezialisierungszeitpunkt partiell ausgewertet (*partial evaluation*) und mit Hilfe der Ergebnisse die Spezialvariante erzeugt, die im Allgemeinen weniger Ressourcen verbraucht.

```
void *memmove(void *dest, const void *src, size_t n) {
   if (src == dest)
      /* tue nichts */
   else if (src < dest && src+n > dest)
      /* kopiere rückwärts */
   else
      /* kopiere vorwärts */
   return dest;
}
```

Abbildung 3.1: Generischer Programmcode von `memmove()`

Zur Verdeutlichung wird als Beispiel eine hypothetische Implementierung der bekannten Funktion `memmove()` aus der C-Befehlsbibliothek betrachtet. Abbildung 3.1 zeigt den dazugehörigen Programmcode. Sie hat die Aufgabe einen Speicherbereich zu kopieren, muss dabei aber den Spezialfall beachten, dass der Quell- und Zielbereich sich überlappen könnten. Wenn nun als Spezialisierungsprädikat `src > dest` angenommenen wird, kann der Code, wie in Abbildung 3.2 auf der nächsten Seite dargestellt ist, vereinfacht werden.

Moderne Spezialisierungswerkzeuge wie Tempo [27] und C-Mix [4] erledigen die entsprechenden Auswertungen und Transformationen automatisch. Dabei sind Spezialisierungen über Funktionsgrenzen hinweg möglich und es werden komplexe Optimierungen, die aus dem Bereich des Übersetzerbaus bekannt sind, wie das Ausrollen von Schleifen (*loop unrolling*) eingesetzt.

Das Konzept der Programmspezialisierung wurde am Beispiel von Betriebssystemcode mehrfach erfolgreich angewendet. Dabei können drei typische Einsatz-

```
void *memmove(void *dest, const void *src, size_t n) {
   /* kopiere vorwärts */
   return dest;
}
```

Abbildung 3.2: Spezialisierter Programmcode von `memmove()`

muster unterschieden werden [79]:

Statische Spezialisierung: Bei der statischen Spezialisierung sind die Spezialisierungsprädikate zur Übersetzungszeit bekannt. Damit kann grundsätzlich die ressourcensparende Spezialvariante eingesetzt werden. In einer Fallstudie konnte damit die Dauer eines entfernten Prozeduraufrufs mit Sun RPC [81] um 55% auf einer Sun/IPX und um 35% auf einer PC/Linux Plattform reduziert werden [82]. Spezialisiert wurde dabei der Code zum Verpacken der Prozedurparameter (*argument marshaling*), der gewöhnlich zusammen mit der Anwendung gebunden wird.

Dynamische Spezialisierung: Wenn die Spezialisierungsprädikate erst zur Laufzeit bekannt sind, muss dynamisch spezialisiert werden. Dazu erfolgt entweder die Codeerzeugung zur Laufzeit oder es wird zur Laufzeit aus einer Menge von Spezialvarianten gewählt. Auf Basis dieser Technik konnte die Auswertung von BSD Paketfiltern [78] deutlich beschleunigt werden.

Optimistische Spezialisierung: Während bei der einfachen dynamischen Spezialisierung davon ausgegangen wird, dass gegen ein einmal festgelegtes Spezialisierungsprädikat nie verstoßen wird, solange der spezialisierte Code in Benutzung ist, wird dies bei der optimistischen Variante zugelassen. Damit dies möglich ist, müssen die Punkte im System identifiziert werden, die zu solchen Verstößen führen können. Durch einzufügenden Testcode wird dafür gesorgt, dass gegebenfalls die spezialisierte Variante wieder durch die generische ersetzt wird. Im einfachsten Fall erfolgt die Umschaltung über Zeiger auf Funktionen. Durch den zusätzlichen Aufwand bei diesem Einsatzmuster lohnt es sich nur, wenn der Code, der für die Ungültigkeit des Prädikats sorgt, selten und der spezialisierte Code oft durchlaufen wird. In einer Fallstudie zur Auslieferung von Signalen in einem Linux System konnte gezeigt werden, dass eine spezialisierte Codevariante unter der Annahme, dass das Prozesspaar aus Signalsender und Signalempfänger gleich dem letzten Prozesspaar ist, sich sehr positiv auf das Laufzeitverhalten bestimmter Anwendungen auswirken kann.

Die Programmspezialisierung ist, insbesondere im statischen Fall, mit der Idee der Programmfamilie verwandt. Bei beiden Ansätzen wird aus generischem Code eine anwendungsspezifische Variante erzeugt. Das Interessante daran ist, dass hier nicht

nur bestimmte Funktionen weggelassen werden können, sondern dass mit Hilfe von Anwendungswissen Ausführungspfade optimiert werden.

Die bisherigen Fallstudien sind im Kontext von Vielzweckbetriebssystemen wie zum Beispiel Linux (siehe oben) oder HP-UX [92] angesiedelt und zielen hauptsächlich auf eine Verbesserung von Laufzeiten ab. Bei der Codegröße sind oftmals Nachteile zu verzeichnen, da beispielsweise die generische und spezialisierte Codevarianten gleichzeitig im Speicher gehalten werden oder zur Laufzeit ein Übersetzer agiert, wie im Fall des Synthesis Betriebssystems [93].

3.3 AOP Anwendungen im Systemkontext

3.3.1 Der a-kernel

Beispiele für die Anwendung von AOP im Betriebssystemsektor sind bisher noch rar. So gibt es eine Fallsstudie, in der exemplarisch die Strategie zum Vorabladen von Datenblöcken im FreeBSD v3.3 System[2] in Form eines Aspekts implementiert wurde [26, 25]. Das Ergebnis dieses Versuchs ist vielversprechend. Die bis dahin über weite Teile des Systems verteilte Implementierung der Vorabladestrategie konnte mit Hilfe eines Aspekts modular und sehr kompakt umgesetzt werden. Dadurch wurde die Implementierung erheblich klarer und eine statische Konfigurierung der Strategie wäre ohne weiteres möglich.

Ein ähnliches Experiment wurde mit der Strategie für das Vorabladen und das verzögerte Schreiben in einem verteilten Dateisystem (NFS) gemacht [24]. Das Ziel dabei war, anders als in üblichen Implementierungen die genannten Strategien in Aspekten zu kapseln, um so eine leichtere Erweiterbarkeit zu erreichen. Auch dieses Experiment wurde positiv beurteilt.

In beiden Fallstudien, die unter dem Oberbegriff **a-kernel** durchgeführt wurden, wurde als Implementierungssprache AspectC verwendet. Es handelt sich dabei um eine aspektorientierte Spracherweiterung für C (nicht C++), die sich an die Semantik von AspectJ anlehnt. Zum Zeitpunkt der Studien gab es keinen Übersetzer für die Sprache, so dass der Code von Hand kompiliert werden musste. Obwohl an einem Übersetzer für AspectC[3] gearbeitet wird, ist bis heute keine Implementierung verfügbar.

Die Stoßrichtung dieser Arbeiten sind Vielzweckbetriebssysteme. Mögliche positive Folgen, die das Konfigurieren von Aspekt- oder Komponentencode in einer Betriebssystemfamilie mit sich bringen könnte, wurden außer Acht gelassen.

3.3.2 Die aspektorientierte Softwarearchitektur für Betriebsysteme AOSA

Bei AOSA handelt es sich um einem Ansatz, Aspekte in Betriebssystemen auch ohne spezielle Sprachunterstützung zu implementieren [85, 84]. Dazu wurde ei-

[2] http://www.freebsd.org/

[3] http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

ne Architektur von geschichteten Komponenten konzipiert und ein Rahmenwerk geschaffen, das Funktionsaufrufe durch einen zentralen "Aspektmoderator" leitet. Aspekte, die in Form normaler C++ Klassen implementiert werden, können sich beim Aspektmoderator anmelden und werden aktiviert, falls eine Funktion, für die sie sich interessieren, abgearbeitet werden soll. Für einige einfache Aspekte wie Kontrollflussverfolgung, gegenseitigen Ausschluss und Synchronisation nach dem Produzenten/Konsumenten Modell konnte gezeigt werden, dass der notwendige Code vom Komponentencode separiert werden konnte.

Offen bleibt bei den dazu bekannten Papieren die Frage, in wie weit sich der zentrale Aspektmoderator und das Abfangen so vieler Funktionsaufrufe negativ auf die Leistung eines Betriebssystems auswirken würden. Darüber hinaus wird auch nur eine sehr einfache Form der aspektorientierten Programmierung unterstützt. Desweiteren ist ohne Sprachunterstützung das explizite Einfügen von Ankerpunkten unvermeidbar. Man könnte daher sogar die Frage stellen, ob es sich hier überhaupt um aspektorientierte Programmierung im Sinne von Abschnitt 2.3 handelt. Positiv zu bemerken ist, dass das Rahmenwerk sehr einfach gestattet, Aspekte zur Laufzeit des Systems hinzuzufügen oder zu entfernen.

3.3.3 Das anwendungsorientierte Betriebssystem EPOS

Eine Vorgängerarbeit aus dem Umfeld der Arbeitsgruppe des Autors beschreibt die anwendungsorientierten Entwurfsprinzipien der EPOS Betriebssystemfamilie [41]. Obwohl der Schwerpunkt dieser Arbeit eher im Bereich der Automatisierung der Systemanpassung und dem Entwurf wiederverwendbarer generativer Komponenten liegt, wurde dort bereits die Nützlichkeit der aspektorientierten Programmierung für Betriebssystemfamilien erkannt. Mit Hilfe sogenannter Szenarioadapter werden in EPOS Systemabstraktionen um wiederverwendbaren Aspektcode erweitert. Ähnlich wie bei dem im letzten Abschnitt beschriebenen AOSA Konzept wurde so beispielsweise die eigentliche Funktionalität von C++ Klassen von Synchronisationscode getrennt, der in einigen Konfigurationen notwendig ist. Zur Implementierung solcher Aspekte wurde aus Mangel an echter sprachlicher Unterstützung C++ *Template*-Metaprogrammierung eingesetzt. Eine Erläuterung und Kritik dieses Ansatzes erfolgt in Abschnitt 3.4.2.

3.3.4 AOP in angrenzenden Bereichen

Im Bereich der verteilten Systeme gibt es bereits mehr Erfahrungen mit der aspektorientierten Programmierung. So wurde schon 1997 mit D [75] ein Rahmenwerk für die verteilte Programmierung mit Java geschaffen, das es erlaubt, in separaten Aspekten zu beschreiben, welche Daten bei Fernaufrufen zu kopieren und welche über entfernte Referenzen anzusprechen sind. Dadurch kann die Datenverteilung und damit auch das Laufzeitverhalten auf einem höheren Abstraktionsniveau implementiert werden.

Daneben existieren Arbeiten zu den Themen Dienstgüte (engl. *quality of service*) als Aspekt [14], Hinzufügen des Verteilungsaspekts mittels CORBA[4] in lokalen Java Code [94] und aspektorientierte *Middleware* [53], um einige Beispiele zu nennen.

Auch eingebettete Systeme zeichnen sich durch die Existenz vieler *Crosscutting Concerns* aus. Dazu gehören die Synchronisation nebenläufiger Aktivitäten, Echtzeiteigenschaften wie Zeitvorgaben durch Sensoren und Aktoren, Fehlertoleranz und mehr. Um diese Systemeigenschaften durch aspektorientierte Programmierung mit weniger Aufwand in den Griff zu bekommen, wurde in den USA das DARPA[5] ITO Projekt Program Composition for Embedded Systems (PCES)[6] aufgesetzt. Es besteht aus 16 Teilprojekten an amerikanischen Universitäten, Forschungs- und Militäreinrichtungen, die alle im Kontext "AOP und eingebettete Systeme" angesiedelt sind. Dazu gehört auch das Projekt Aspect-Oriented Programming im Xerox PARC[7], wo der Begriff AOP geprägt und AspectJ entwickelt wurde.

3.4 Aspektsprachen zur Systemprogrammierung

Die Menge der existierenden Programmiersprachen mit Unterstützung für AOP beschränkt sich fast ausschließlich auf Erweiterungen zu Java oder Smalltalk[8]. Da diese Sprachen typischerweise nach Übersetzung in einen *Byte Code* durch eine virtuelle Maschine interpretiert oder zur Laufzeit in Maschinencode übersetzt werden, benötigen sie ein großes Laufzeitsystem, was die Benutzung im Kontext kleinster eingebetteter Systeme aussschließt. Eine Betriebssystemfamilie würde so diesen sehr bedeutenden Anwendungsbereich verlieren, so dass diese Sprachen für die Entwicklungen, die in dieser Arbeit beschrieben werden, nicht in Frage kommen.

AOP Erweiterungen für vollständig in Maschinencode übersetzte Sprachen wie C oder C++ sind bisher rar. Neben dem bereits angsprochenen AspectC exisitieren hier lediglich zwei weitere Ansätze, die in den folgenden Abschnitten vorgestellt werden.

3.4.1 FOG

Der Flexible Object Generator (FOG) [115][114] ist ein Präprozessor für C++. Er kann als (nicht kompatibler) Ersatz für den von C übernommenen Präprozessor `cpp` eingesetzt werden und hat darüber hinaus Eigenschaften, die auch die Implementierung von Aspekten gestatten. So wird mit FOG die sogenannte *One Definition*

[4] http://www.omg.org/

[5] Die Defense Advanced Research Projects Agency (DARPA) fördert Forschung für das amerikanische Verteidigungsministerium (DoD)

[6] http://www.darpa.mil/ito/research/pces/

[7] Palo Alto Research Center

[8] Eine Übersicht findet man unter http://www.aosd.net/

Rule von C++ außer Kraft gesetzt, die besagt, dass jedes Objekt und jede Funktion nur einmal definiert werden dürfen. Stattdessen werden mehrfache Definitionen verbunden, sofern sie sich nicht widersprechen. Das folgende Beispiel zeigt, wie auf diese Weise Klassen aus Stücken zusammengesetzt werden können.

```
class One
 {
   class Inner {}; // eine normale innere Klasse
   class ::Two     // keine innere Klasse!
    { void f (); };
 };

class Two
 { void g (); };    // erweitert die Klasse 'Two'

private bool Two::_attribute = false; // noch eine Erweiterung
```

Die Klasse `Two` wird hier um die Methoden `f()` und `g()` sowie um das Attribut `_attribute` erweitert. Wie zu erkennen ist, mussten dazu einige Erweiterungen in der C++ Syntax eingeführt werden.

In analoger Weise kann der Code von Funktionen erweitert werden. Die Rümpfe mehrfacher Funktionsdefinitionen werden einfach verbunden. Spezielle Schlüsselworte erlauben auch das gezielte Einfügen von Code am Anfang oder am Ende:

```
class TargetClass
 {
   public void work()
   :{
      entry { do_before (); };
      exit { do_after (); };
    };
 };
...
class TargetClass
 {
   public:
     void work ()
      { do_something (); }
 };
```

FOG würde die durch einen Aspekt beigesteuerte obere Klassendefinition mit der unteren verbinden. Das Resultat entspricht etwa dem `before` und `after` *Advice* von AspectJ (siehe Abschnitt 2.3).

Die Möglichkeiten von FOG gehen weit über die hier präsentierten Beispiele hinaus. Es wird ein sogenannter *Compile-Time Meta Level* geschaffen, womit komplexe Programm-Manipulationen zur Übersetzungszeit möglich sind. Damit verfolgt FOG den gleichen Weg wie bereits 1995 OpenC++ [22], nur dass FOG dabei auf eine Spracherweiterung setzt, während bei OpenC++ die Manipulationen in einem separaten Modul kompiliert werden.

FOG eignet sich hervorragend für das Zusammensetzen von Klassen aus verschiedenen Quellen, die beispielsweise verschiedene Sichtweisen auf das gleiche Subjekt repräsentieren. Diese Art der Programmierung wird unter dem Begriff *Subject-Oriented Programming* [52] propagiert. Was FOG dagegen fehlt, ist ein Konstrukt wie der *Pointcut* in AspectJ. Ein *Pointcut* kann Mengen von Verbindungspunkten flexibel beschreiben. Solange dies nicht gegeben ist, kann FOG nur als bedingt nützlich für die aspektorientierte Programmierung mit C++ angesehen werden.

3.4.2 AOP mit C++ *Template*-Metaprogrammierung

Die C++ *Template* Metaprogrammierung ermöglicht die Ausführung von Code zur Übersetzungszeit unter Ausnutzung der sogenannten *Template* Spezialisierung. Das folgende Beispiel zur Berechnung der Fakultät einer beliebigen Konstanten soll dies verdeutlichen:

```
template<int n>
struct Factorial
 { enum { RET = Factorial<n-1>::RET * n };
 };

template<>
struct Factorial<0>
 { enum { RET = 1 };
 }
```

Das *Template* `Factorial` wurde hier in einer allgemeinen Ausprägung und mit einer Spezialisierung für den Argumentwert 0 definiert. Da der C++ Übersetzer Konstanten und *Templates* zur Übersetzungszeit auswerten kann, ist der generierte Code für die beiden folgenden Anweisungen identisch.

```
cout << Factorial<7>::RET << endl;
cout << 5040 << endl;
```

Man kann diese Spracheigenschaft ausnutzen, um so auch Kontrollkonstrukte wie ein IF *Template* zu erstellen. Folgt man diesem Weg weiter, so erhält man eine funktionale Programmiersprache, deren zur Übersetzungszeit ausgewertete Programme die Codegenerierung erheblich beeinflussen können. Es liegt nahe, zu untersuchen, ob damit das Binden von Aspekten an C++ Klassen ermöglicht werden kann.

Ein erster dazu publizierter Ansatz [31] beruht auf der Idee, C++ Namenräume, Weiterleitung, Vererbung und *Mixins* (siehe Abschnitt 2.2.2) in Kombination zu nutzen. Das Codebeispiel in Abbildung 3.3 auf der nächsten Seite zeigt, wie man damit einen Aspekt auf eine Klasse wirken lassen kann.

Durch die Verwendung von Namensräumen können `Components::Worker` und `Composed::Worker` koexisitieren. Code, der die zusammengesetzte Klasse nutzen soll, muss lediglich um `using namespace Composed;` erweitert werden. Die

```
namespace Components
 {
   class Worker
    { //...
      void work () { /* berechne etwas */ };
    };
   // weitere Klassen
 }

namespace Aspects
 {
   template<class Component>
   class Aspect : public Component
    {
      void work ()
       {
         do_before ();
         Component::work ();
         do_after ();
       }
    };
 }

namespace Composed
 {
   // hier wird der Aspekt an die Klasse gebunden
   typedef Aspects::Aspect<Components::Worker> Worker;
 }
```

Abbildung 3.3: Aspekte mit Hilfe von *Templates*

Aspektimplementierung erbt von der als *Template*-Parameter übergebenen Klasse `Component`. Damit sind all ihre Operationen und Daten auch in der zusammengesetzten Klasse verfügbar. Durch das Überdefinieren einzelner Methoden mit Weiterleitung an die entsprechende Basisklassenmethode, kann Code vor oder nach Ausführung der Originalmethode eingefügt werden. Das ist mit *Before*- und *After-Advice* von AspectJ vergleichbar.

Komplizierter wird es, wenn mehrere Aspekte in beliebiger Reihenfolge auf Klassen wirken sollen, wenn Konstruktoren Parameter haben, wenn Code vor oder nach der Ausführung von Konstruktoren oder Destruktoren ausgeführt werden soll oder wenn in den Signaturen der Methoden Datentypen aus der Originalklasse auftauchen. In all diesen Fällen ist erheblich mehr Code oder auch die Implementierung eines *Template*-Metaprogramms erforderlich. Es scheint nicht möglich zu sein, ein Rahmenwerk von AOP *Templates* zu schaffen, mit dem das Binden eines beliebigen Aspekts an beliebige Klassen einfach zu bewerkstelligen ist.

Neben diesem Problem fällt auch auf, dass im Vergleich zu AspectJ wesentliche Eigenschaften fehlen. Dazu gehört vor allem, dass für die Konzepte der *Pointcuts* und GTNs, mit denen beliebige Mengen von Verbindungspunkten oder Klassen flexibel beschrieben werden können, kein Äquivalent existiert. Verwendet werden diese Konzepte damit *Advice*-Code bzw. *Introductions* auf eine ganze Menge von Verbindungspunkten im System wirken können und nicht nur auf eine einzelne Klasse. Die Implementierung eines generischen Aspekts zur Kontrollflussverfolgung würde daher wohl scheitern.

Aufgrund dieser Mängel und auch wegen der Kompliziertheit solcher Programme sollte AOP mit *Template*-Metaprogrammierung nur als Notlösung angesehen werden, falls keine adäquate programmiersprachliche Unterstützung zur Verfügung steht.

3.5 Zusammenfassung

Zusammenfassend lässt sich zum Stand der Kunst im Bereich "Aspektorientierung und Programmfamilien im Betriebssystembau" Folgendes sagen: Programmfamilien werden heute durch den großen und populären Forschungsbereich der Produktlinien abgedeckt. Arbeiten, die einen Bezug von Produktlinien zu AOP herstellen, existieren. Es handelt sich aber um nur wenige. Gleiches gilt für Arbeiten, die eine Verbindung zwischen Produktlinien und Systemsoftware herstellen.

Die ersten Papiere zum Thema Aspektorientierung und Betriebssysteme stellen vielversprechende Fallstudien dar. Es handelt sich aber um zu wenige Erfahrungen, um absehen zu können, welche Vorteile sich in dem durch viele *Crosscutting Concerns* charakterisierten Bereich der Betriebssysteme durch AOP ergeben könnten.

Damit tut sich hier ein lohnenswertes Betätigungsfeld für praktische Forschung auf, bei dem man, mindestens wenn es um die Kombination von AOP, Programmfamilien und Betriebssysteme geht, absolutes Neuland betritt.

Leider werden die notwendigen Untersuchungen und Experimente dadurch

gebremst, dass eine adäquate Programmiersprache, die AOP unterstützt, zu verständlichem und somit wartbarem Code führt und gleichzeitig für die Entwicklung schlanker Systemsoftware für den Bereich kleinster eingebetteter Systeme geeignet ist, nicht existiert.

Kapitel 4

Problemanalyse und Lösungsansatz

Betriebssystemfamilien zu entwickeln, die für ein gegebenes Anwendungsszenario einen minimalen Ressourcenverbrauch aufweisen, ist eine schwierige Aufgabe. Die Gründe dafür sollen im Verlauf dieses Kapitels analysiert werden. Dazu wird zuerst ganz allgemein der Konflikt zwischen Wiederverwendung und Effizienz beschrieben und erläutert, warum konfigurierbare Software hilft, diesen Konflikt zu lösen. Als Beispiel wird dargestellt, wie die Konfigurierung von PURE technisch umgesetzt wird und welche Ergebnisse bisher dort vorzuweisen sind. Dann werden die in der Einleitung grob skizzierten Probleme bei dem Entwurf und der Implementierung von Betriebssystemfamilien aufgegriffen und detaillierter dargestellt. Diese Problemanalyse führt zu einem Lösungsansatz, der viele Konfigurierungsmaßnahmen unnötig macht, wodurch einerseits die Entwicklung und Wartung erleichtert werden und andererseits Systemeigenschaften konfigurierbar werden, deren Konfigurierung bisher nur mit sehr großem Aufwand möglich war.

4.1 Hintergrund

4.1.1 Wiederverwendung versus Effizienz

Wie in der Einleitung bereits angesprochen wurde, müssen Entwickler von Betriebssystemfamilien, beispielsweise im Bereich kleinster eingebetteter Systeme, von Ressourcenknappheit ausgehen. Dies führt zu besonderen Maßnahmen in vielen Phasen des Entwicklungsprozesses. Eine dieser Maßnahmen ist die anwendungsspezifische Anpassung von verwendeten Bibliotheken. Ein Betriebssystem wie PURE kann als Spezialfall einer solchen Bibliothek angesehen werden. Der Gewinn an eingesparter Codegröße, der durch Anpassung erzielt werden kann, ist erheblich. So verringert sich beispielsweise die Größe eines in C geschriebenen "hello, world" Programms unter Linux/x86 von etwa 350 KBytes auf 157 Bytes, wenn man statt der mächtigen Funktion `printf` für die Ausgabe den Systemdienst

`write` direkt benutzt[1]. Das heißt, eine für diese Anwendung angepasste C Bibliothek könnte die gleiche Leistung mit nur 0.05 % des Speicherplatzverbrauchs erbringen.

Anzunehmen ist auch eine Reduzierung der Ausführungszeiten und damit eine Verbesserung der Rechenleistung. Dies ist beispielsweise der Fall, wenn das verwendete Rechnersystem einen Speicher-*Cache* aufweist, der durch die Verkleinerung des Codes nun einen größeren Teil des Softwaresystems abdeckt. Im Bereich der kleinsten eingebetteten Systeme, wo hauptsächlich 4 und 8 Bit Mikrocontroller eingesetzt werden, sind Speicher-*Caches* jedoch nicht die Regel. Somit wird dieser Effekt selten eintreten. Für wirkliche Laufzeitersparnisse ist es erforderlich, nicht nur unbenutzten Code wegzulassen, sondern es muss der tatsächlich durchlaufene Code beeinflusst werden. So können beispielsweise in bestimmten Anwendungsszenarien Fallunterscheidungen zur Laufzeit entfallen, wenn das Ergebnis bereits zur Übersetzungszeit bestimmt werden kann. Dies ist zum Beispiel in dem "hello, world" Beispiel der Fall.

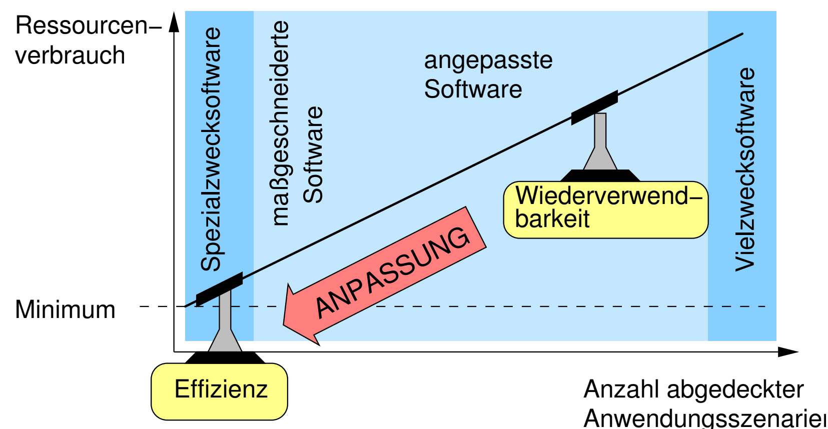


Abbildung 4.1: Konflikt zwischen Effizienz und Wiederverwendbarkeit

Die Obergrenze des Ressoucensparens ist bei den genannten Maßnahmen durch den Ressoucenverbrauch einer Spezialzweckimplementierung für das jeweilige Anwendungsszenario definiert. Abbildung 4.1 verdeutlicht diesen Zusammenhang. Dort wird schematisch der Ressourcenverbrauch eines Softwaresystems für einen bestimmten Anwendungsfall gezeigt. Im Fall einer Spezialzwecksoftware, die nur im Hinblick auf dieses Szenario erstellt wurde, kann der Ressourcenverbrauch bei einer guten Implementierung dicht am Minimum liegen. Vielzwecksoftware dagegen verbraucht im gleichen Anwendungsszenario deutlich mehr Ressourcen, da sie unbenutzten Code enthält und unnötige Fallunterscheidungen zur Laufzeit durchführt. Im mittleren Bereich ist dargestellt, dass durch Anpassung

[1] Der Effekt tritt nur ein, wenn der Binder angewiesen wird, den Linux-spezifischen Startup-Code wegzulassen (`g++ -nostartfiles`), da dieser ebenfalls (indirekt) `printf` referenziert.

von wiederverwendbaren Bausteinen der Ressourcenverbrauch im Vergleich zur Vielzwecksoftware gesenkt werden kann. Im Idealfall führt die Anpassung zur "maßgeschneiderten Software", das heißt Software, die auf das jeweilige Anwendungsszenario so genau abgestimmt ist, dass der Ressourcenverbrauch an den der Speziallösung heranreicht. Auf diese Weise wird durch Anpassung der Konflikt zwischen Effizienz und Wiederverwendung gelöst.

4.1.2 Statische und dynamische Konfigurierungsmaßnahmen

Die Anpassung einer Software kann sowohl statisch zur Übersetzungs- oder Bindungszeit als auch dynamisch zur Laufzeit geschehen. Einfache statische Konfigurierungsmaßnahmen sind die bedingte Übersetzung, die durch Konstrukte der Programmiersprache oder mittels eines Präprozessors ermöglicht wird, und die Selektion aller von der Anwendung referenzierten Module einer Bibliothek durch den Binder. Komplexe statische Konfigurierungsmaßnamen sind globale Optimierungen wie die partielle Evaluation (siehe Abschnitt 3.2). Spezialisierungswerkzeuge wie Tempo oder C-Mix manipulieren auf Basis relativ weniger Informationen ein Softwaresystem an vielen unterschiedlichen Stellen. Dies ist eine deutliche Parallele zu Aspektwebern. Da sich die partielle Evaluation im Wesentlichen an der Programmiersprache der manipulierten Software orientiert und wenig externes Wissen über das Anwendungsszenario nutzt, ist sie nicht in der Lage alle Aspekte der Systemkonfigurierung abzudecken. Es bietet sich daher an, ein System zunächst auf Basis der einfachen statischen Konfigurationsmechanismen grob zuzuschneiden und anschließend den resultierenden Code noch globalen Optimierungen zu unterziehen, falls entsprechende Werkzeuge vorhanden sind.

Die dynamische Rekonfigurierung eines Systems zur Laufzeit eröffnet weitere Optimierungspotentiale. So können zeitweise unbenutzte Teile des Systems aus dem Speicher entfernt werden. Im Extremfall bleibt nur noch ein minimales *Working Set* übrig, durch das gewährleistet wird, dass nicht zu oft Teile des Systems nachgeladen werden müssen. Daneben existieren auch dynamische Spezialisierungstechniken, mit deren Hilfe Ausführungspfade zur Laufzeit an die Erfordernisse des aktuellen Anwendungsverhaltens angepasst werden. Die in Abschnitt 3.2 vorgestellten Beispiele haben gezeigt, dass solche Techniken erfolgreich angewendet werden können. Alle dynamisch rekonfigurierbaren Softwaresysteme haben jedoch die gemeinsame Eigenschaft, dass ein Laufzeitsystem benötigt wird, um Ein- und Auslagerungs- bzw. Spezialisierungsentscheidungen zu treffen und um die Rekonfigurierung an sich durchzuführen. Diese Laufzeitumgebung ist in ihrem Ressourcenverbrauch fest. Ob es sich lohnt, solche dynamischen Techniken einzusetzen, hängt davon ab, wie das Verhältnis zwischen dem Gesamtressourcenverbrauch und dem Ressourcenverbrauch der Rekonfigurierungsinfrastruktur ist.

In der für Betriebssystemfamilien wichtigen Domäne der kleinsten eingebetteten Systeme ist eine Rekonfigurierungsinfrastruktur in der Regel nicht einsetzbar. Dafür gibt es mehrere Gründe:

- Der Speicherplatzbedarf für die Infrastruktur ist zu groß.
- Typischerweise existiert keine Kommunikationsverbindung bzw. kein größeres Speichermedium, um neu benötigte Teile des Systems nachzuladen.
- Das Anwendungsprofil ändert sich viel weniger als in Vielzwecksystemen wie PCs.

Trotzdem bleiben diese Techniken interessant und Lösungen für die genannten Probleme sind denkbar. So könnte viel der Infrastruktur auf leistungsstärkere Rechner ausgelagert werden, falls eine Kommunikationsverbindung exisitiert [15]. Solche Ansätze sind Gegenstand aktueller und zukünftiger Forschungen. Der logische Weg besteht jedoch darin, zunächst die in dieser Domäne wichtigere statische Konfigurierung zu beherrschen. Dazu gehören sowohl Konfigurierungstechniken als auch Modelle zur Verwaltung von Konfigurierungsinformationen, die durch eine durchgängige Methode verbunden werden.

4.1.3 Beispiel: PURE

Da die Zieldomäne kleinste eingebette Systeme sind, wird bei PURE bisher ausschließlich die statische Konfigurierung verwendet. Der Entwurf des Systems beruht auf dem Prinzip der Programmfamilie (siehe 2.2), während die Implementierung dieses Konzept objektorientiert umsetzt [95]. Der Grund dafür besteht in der Dualität zwischen den Konzepten der minimalen Erweiterung und der Vererbung, die in Abbildung 4.2 angedeutet wird [97].

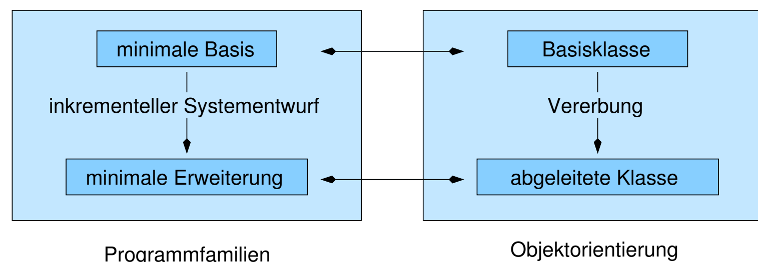


Abbildung 4.2: Dualität von minimalen Erweiterungen einer Programmfamilie und objektorientierter Vererbung

Der Vorteil dieser Art der Implementierung besteht darin, dass die funktionale Hierarchie des familienbasierten Entwurfs direkt in eine Klassenstruktur umgesetzt werden kann und dass der Binder einen Großteil der Konfigurierung bereits erledigt. Dies geschieht dadurch, dass eine PURE Anwendung nur die Klassen zur Objektinstanziierung nutzt, die ihre Anforderungen gerade eben erfüllen und somit keine unnötige Funktionalität bieten. In vielen Fällen werden also Klassen aus der Mitte einer Vererbungshierarchie verwendet. Da PURE als Bibliothek vorliegt,

bedeutet dies wiederum, dass der Code der weiter spezialisierten, das heißt abgeleiteten Klassen nicht referenziert und damit durch den Binder nicht in das fertige System übernommen wird.

Eine zweite Dimension der Konfigurierbarkeit entsteht bei PURE durch die Verwendung von konfigurierbaren Klassen, die auch als Klassenaliase bezeichnet werden. Technisch gesehen handelt es sich dabei um eine Typdefinition (`typedef`), die wahlweise auf eine von mehreren alternativen Klassen verweist. Ursprünglich erfolgte diese Konfigurierung durch bedingte Übersetzung. Heute werden die Typdefinitionen mit Hilfe eines Konfigurationswerkzeugs generiert (siehe 4.2.1).

Abbildung 4.3 auf der nächsten Seite zeigt als Beispiel für die Umsetzung des beschriebenen Konzepts einen Ausschnitt aus der Klassenhierarchie von PURE. Es handelt sich dabei um Klassen, die die nötigen Daten für verschiedene Fadenabstraktionen und die notwendigen Methoden bereitstellen. Die mehrere Klassen umschließenden Blöcke *Trimmer*, *Temper* und *Informer* sind Klassenaliase. Sie können je nach Konfigurierung in Form einer der eingeschlossenen Klassen in Erscheinung treten. *Schemer* ist ebenfalls ein Klassenalias. Auf die Darstellung der vielen dort möglichen Varianten wurde aus Platzgründen verzichtet.

Es fällt auf, dass sehr viele Klassen benötigt werden, um eine relativ kleine Datenstruktur und ihre Methoden zu implementieren. Das deutet darauf hin, dass jede einzelne Klasse nur einen minimalen Funktionsumfang hat, was ein Blick in den Quelltext bestätigen würde. Der Grund dafür ist das Konzept der "minimalen" Erweiterungen der funktionalen Hierarchie. Dieses Konzept wird verfolgt, um feingranulare Konfigurierbarkeit zu erzielen. Durch die direkte Abbildung von Funktionen der Familie auf Klassen überträgt sich diese Minimalität auf die Implementierungsstruktur.

Konfiguration	Größe von Code+Daten (Bytes)	Kontextwechsel (Taktzyklen)
Einzelnes aktives Objekt	434	–
Kooperatives Scheduling	1648	61
Präemptives Scheduling	4062	368

Tabelle 4.1: Größe und Kontextwechseldauer verschiedener PURE Konfigurationen

Die Ergebnisse der PURE Entwicklung sind überzeugend. Tabelle 4.1 zeigt beispielsweise die Codegrößen und die Dauer eines Kontextwechsels für einige ausgewählte PURE Konfigurationen[2] [17]. Die Konfiguration "Einzelnes aktives Objekt" besteht lediglich aus dem notwendigen Initialisierungscode für die Rechnerhardware und den notwendigen Klassen, um der Anwendung die Abstraktion aktiver Objekte zur Verfügung zu stellen. Die Verwendung mehrerer kooperativer aktiver Objekte, vergleichbar mit dem Konzept der Coroutinen, wird durch die zweite

[2] Die Zahlen beziehen sich auf die PURE Konfiguration für x86 CPUs. Sie wurde mit dem egcs-1.0.2 übersetzt. Die Messung der Laufzeiten erfolgte mit einem Pentium II Prozessor.

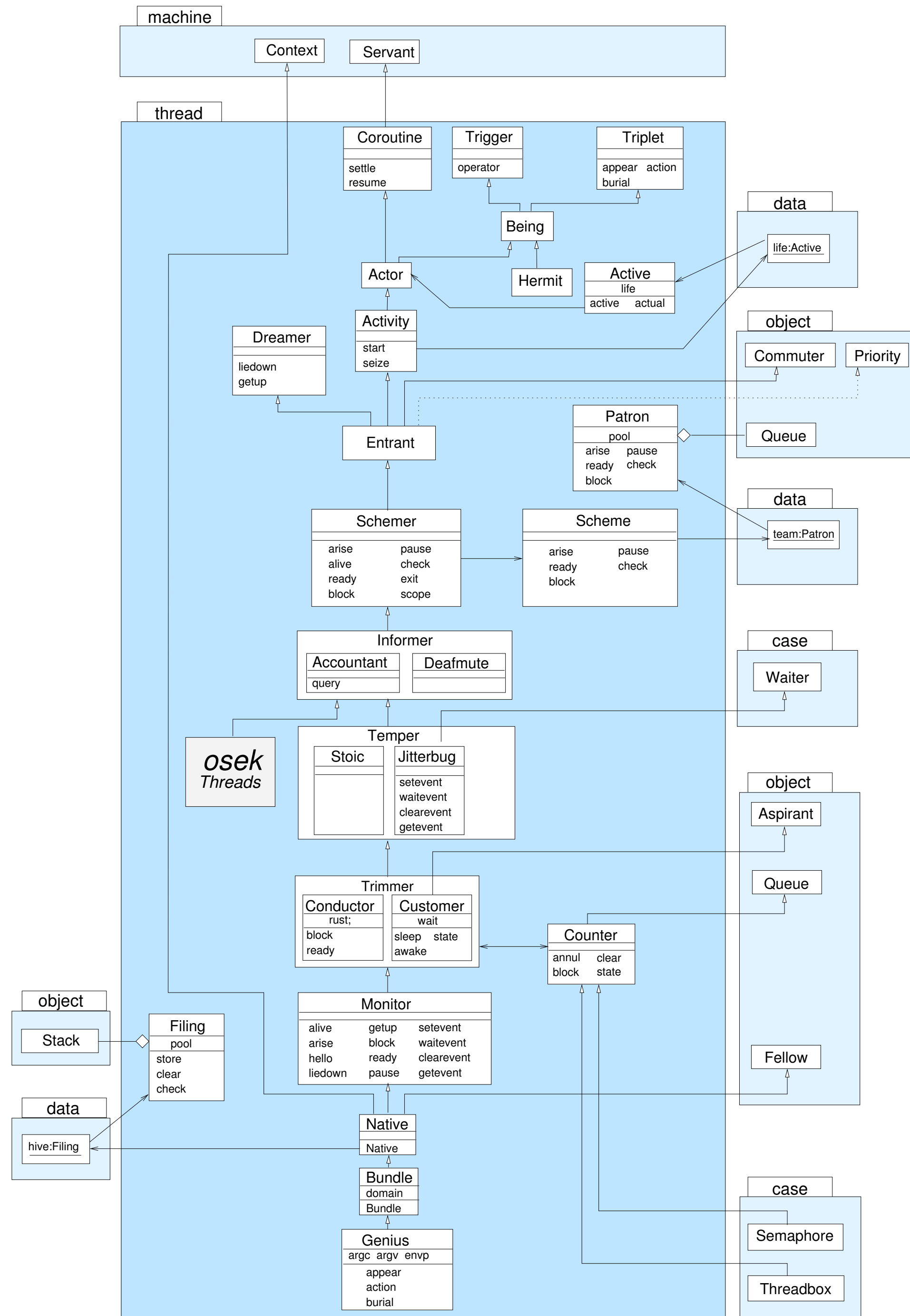


Abbildung 4.3: Die PURE Fadenhierarchie

Konfiguration bereitgestellt. In der dritten Beispielkonfiguration erfolgt unterbrechungsgetriebenes präemptives Scheduling von aktiven Objekten. Die absoluten Zahlen zeigen bereits, dass mit PURE sehr schlanke und effiziente Spezialzwecksysteme erstellt werden können. Vor allem wichtig ist dabei, dass die Größen und Laufzeiten in Abhängigkeit von der Systemkonfiguration und damit von den Anwendungsanforderungen skalieren. Dass PURE dieses Kriterium erfüllt, wird auch durch detailliertere Messungen bestätigt [16]. Dies gilt nicht nur für die Zahlen der hier betrachteten x86 Variante von PURE, sondern auch für kleine 8 Bit Systeme, die das eigentliche Ziel der PURE Entwicklung sind. So hat ein Vergleich mit anderen Betriebssystemen für 8 Bit Atmel AVR Mikrocontroller gezeigt, dass PURE auch hier in vielen Anwendungsfällen durch seine Skalierbarkeit der Konkurrenz in puncto Speicherverbrauch zum Teil erheblich überlegen ist [43].

4.2 Problemdiskussion

Das Fazit der Betrachtungen des vorangegangenen Abschnitts ist, dass bereits heute die Entwicklung von feingranular konfigurierbaren Programmfamilien möglich ist und dass die gewünschte Skalierbarkeit des Ressourcenverbrauchs tatsächlich eintritt. Es geht nun darum, die Methoden und Werkzeuge, die für die Erstellung einer solchen Familie benutzt werden können, kritisch zu untersuchen. Dazu werden die aufgetretenen Probleme bei dem Entwurf und der Implementierung der Betriebssystemfamilie PURE detailliert erörtert. Dabei steht PURE hier repräsentativ für die Menge der statisch konfigurierbaren Softwaresysteme, deren Ressourcenverbrauch durch die Konfigurierung in Abhängigkeit von den Anforderungen skalierbar sein soll. Die Ergebnisse sind daher übertragbar. Das spätere Ziel ist es, durch die Lösung dieser Probleme allgemein die Entwicklung von Programmfamilien zu erleichtern.

4.2.1 Konfigurationsvielfalt

Die Erfahrungen mit PURE haben gezeigt, dass funktionale Hierarchien, bei denen jede Ebene eine minimale Erweiterung der darunter liegenden Ebene ist, gut geeignet sind, um auch feingranular konfigurierbare Programmfamilien zu entwerfen. Wichtig und nützlich war dabei die Anwendung des Konzepts der schnittstellenkompatiblen alternativen Funktionen, das den austauschbaren Schichten im GenVoca Modell entspricht.

Je feingranularer ein System konfigurierbar ist, desto besser lässt es sich an das jeweilige Anwendungsszenario anpassen. Bei PURE geht diese Konfigurierbarkeit bis hinunter auf die Ebene einzelner Funktionen oder Attributdefinitionen in Klassen. Dabei hat sich herausgestellt, dass in derartig variantenreichen Systemen die Konfigurationsvielfalt ohne die Unterstützung durch spezielle Methoden und Werkzeuge kaum beherrscht werden kann. Dies gilt allein schon deshalb, weil nicht alle Kombinationen sinnvoll oder überhaupt übersetzbar sind. Auf der Ent-

wurfsebene muss daher neben der funktionalen Hierarchie, bzw. GenVoca Grammatik, auch eine Modellierung der tatsächlich sinnvollen Konfigurationen erfolgen, die über die Frage der Schnittstellenkompatibilität hinausgeht.

Die sogenannten *Design Rules* sind ein Konzept für die Modellierung solcher Abhängigkeiten [9]. Sie wurden als Teil des GenVoca Modells konzipiert und erweitern die Komponentenbeschreibungen um Attribute und Prädikate, wodurch automatisch überprüfbare Beschränkungen der Konfigurationsvielfalt beschrieben werden können. Der Nachteil dieses an sich schlagkräftigen Ansatzes wie auch des gesamten GenVoca Konzepts ist, dass die Konfigurierung durch einen Experten durchgeführt werden muss, der die Grammatik und damit die Modularisierung des konfigurierten Systems kennt. Schließlich müssen im Falle von Fehlermeldungen aus der *Design Rule* Überprüfung die Gründe erkannt werden und eine neue sinnvolle Konfiguration muss gefunden werden.

Um auch Entwicklern die Konfigurierung eines intern unbekannten Systems zu erlauben, wurde im Kontext von PURE ein anderer Weg beschritten. Mit Hilfe der in Abschnitt 2.2.3 bereits beschriebenen Feature Modelle erfolgt vor dem Entwurf der funktionalen Hierarchie eine Analyse der möglichen Systemeigenschaften der Familie und deren Beziehungen. Dieser Schritt hilft dabei, sich über notwendige Konfigurierungsmaßnahmen und Schichten im Klaren zu werden, und kann gleichzeitig bei der Systemkonfigurierung genutzt werden. Mit einem Feature-Modell erfolgt die Konfigurierung auf Basis von Systemeigenschaften, die mit der Modularisierung nichts zu tun haben. Dadurch ist für die Konfigurierung kein Wissen über die Modularisierung erforderlich. Nach der Auswahl der gewünschten Eigenschaften erfolgt mit Hilfe einer Wissensbasis eine Abbildung der Eigenschaften auf notwendige Systemkomponenten und Konfigurierungsschalter, um genau diese Eigenschaften und möglichst nicht mehr zu erbringen. Die Wissensbasis muss von den Entwicklern des konfigurierbaren Systems erstellt werden. Mit Hilfe eines Werkzeugs namens **Consulat** wird sowohl der Entwicklungsprozess, das heißt der Entwurf des Feature Modells und die Entwicklung der Abbildungsregeln, als auch die Systemkonfigurierung, das heißt die Auswahl von Eigenschaften, deren Abbildung und die Konfigurierungsmaßnahmen, unterstützt.

Mit der beschriebenen Lösung ist die Forschung in diesem Bereich sicher nicht abgeschlossen. Es hat sich beispielsweise gezeigt, dass Feature Modelle vielfach nicht ausreichen, um adäquate Beschreibungen der Variabilität einer Programmfamilie zu erstellen. Hier notwendige Erweiterungen und viele andere Fragestellungen sind Thema eines parallel laufenden Promotionsvorhabens[3]. Daher wird das Thema Verwaltung von Konfigurationsinformation in den folgenden Abschnitten und Kapiteln nicht weiter betrachtet.

[3]Danilo Beuche, Otto-von-Guericke-Universität Magdeburg und pure-systems GmbH

4.2.2 Crosscutting Concerns

In Abschnitt 4.1.3 wurde beschrieben, wie bei der Entwicklung von PURE die Funktionen der funktionalen Hierarchie eines familienbasierten Systementwurfs auf Implementierungsmodule (hier Klassen) abgebildet werden. Nur durch diese direkte Abbildung können die klaren Abhängigkeitsbeziehungen für eine einfache Konfigurierung ausgenutzt werden.

Ein *Crosscutting Concern* passt nicht in dieses einfache Schema. Es lässt sich mit herkömmlichen Techniken nicht in Form eines Moduls implementieren. Häufig resultieren *Crosscutting Concerns* aus nicht-funktionalen Systemanforderungen, also dem "wie" der Implementierung. Es ist damit nicht überraschend, dass sie in einer "funktionalen" Hierarchie nicht berücksichtigt werden. Damit ist neben der Implementierung auch der Entwurf einer Programmfamilie, in der *Crosscutting Concerns* eine Rolle spielen, zu überdenken.

Ein Ansatz zur Lösung dieser entwurfsseitigen Probleme exisitiert im GenVoca Modell (siehe 2.2.2). Durch das Attributieren von Schichtimplementierungen mit nicht-funktionalen Eigenschaften besteht bei der Auswahl eines Familienmitglieds die Möglichkeit, solche Eigenschaften zu berücksichtigen und bespielsweise die gleiche Funktion wahlweise speicherplatz- oder rechenzeitsparend zu erbringen. Dies setzt jedoch voraus, dass entsprechende Implementierungsvarianten bereits vorliegen, das heißt, in der Grammatik erfasst sind.

Wie in Abschnitt 2.3 beschrieben wurde, können *Crosscutting Concerns* bei entsprechender Sprachunterstützung auch als Aspekte implementiert werden. Aspekte sind Implementierungsmodule, die auch nicht-funktionale Eigenschaften erbringen können. Mit der indirekten Art der Modellierung nicht-funktionaler Eigenschaften im GenVoca Modell ist das nicht zu vereinbaren. Gleichzeitig können Aspekte wegen ihrer völlig anderen Art mit anderen Modulen zu interagieren auch nicht als Schichtimplementierungen angesehen werden. Damit finden sie keinen Platz in der GenVoca Grammatik. Um die Struktur einer Programmfamilie zu entwerfen, bei der *Crosscutting Concerns* mittels Aspekten ausgedrückt werden, muss daher eine Erweiterung des Modells in Betracht gezogen werden.

Bei der Modellierung der Eigenschaften und der Konfigurierung des Systems mittels Feature Modellen oder auch *Design Spaces* stellen Aspekte dagegen kein Hindernis dar. In diesen Modellen besteht kein Unterschied zwischen einem Feature "Kontrollflussverfolgung" und einem Feature "VFAT-Dateisystem". Dabei ist die Kontrollflussverfolgung ein *Crosscutting Concern* und die Unterstützung eines bestimmten Dateisystemtyps nicht. Die Unterscheidung dieser beiden Feature-Arten wird erst beim Übergang zum Entwurf einer Modulstruktur relevant.

4.2.3 Auswahl der Implementierungsstruktur

Ein wichtiger Schritt bei der Implementierung eines als funktionale Hierarchie entworfenen Systems ist die Auswahl einer geeigneten Implementierungsstruktur. Eines der Probleme, die dabei auftreten können, soll nun vorgestellt werden. Dabei

kommt es zu einer kombinatorischen “Explosion” der Anzahl der Klassen im System, wenn sich benutzende Klassen jeweils in mehreren Varianten vorliegen und diese Klassen unter Verwendung von Vererbung miteinander kombiniert werden sollen. Als Beispiel soll hier die Beziehung von Dateisystemen und Gerätetreibern betrachtet werden.

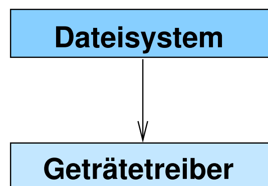


Abbildung 4.4: Abhängigkeitsbeziehung zwischen Dateisystemen und Gerätetreibern

Dateisysteme benutzen für den Zugriff auf das physikalische Speichermedium einen Gerätetreiber. Diese Beziehung der entsprechenden Systemfunktionen ist in der funktionalen Hierarchie in Abbildung 4.4 zu sehen. Für beide Schichten sind mehrere Varianten denkbar. So könnte die Familie etwa die Dateisysteme FAT und ISO9660 sowie Gerätetreiber für Disketten, Festplatten und CDROMs umfassen. Schon bei dieser ausgesprochen grobgranularen Konfigurierung werden sechs Kombinationen erreicht.

In PURE wird dieses Problem in den meisten Fällen dadurch gelöst, dass die Varianten zu Ausprägungen einer konfigurierbaren Klasse werden. Diese Lösung bedeutet allerdings, dass zum Konfigurationszeitpunkt genau eine der vielen möglichen Kombinationen gewählt wird. Ein Nebeneinander von mehreren dieser Kombinationen ist nicht verfolgt worden. Für das Beispiel bedeutet das, dass etwa das FAT Dateisystem als Erweiterung des Festplattengerätetreibers angesehen würde und nicht gleichzeitig noch für Disketten zur Verfügung stehen könnte. Diese Vorgehensweise mag möglicherweise zunächst einmal inakzeptabel erscheinen, ist für viele Anwendungsszenarien jedoch ausreichend.

Ein Kompromiss zwischen dem statischen Entscheiden für genau eine Kombination und dem Bereitstellen aller Kombinationen könnte darin bestehen, die Klassen der verschiedenen Ebenen mit Hilfe von generischem Code im Bedarfsfall zu generieren. Hierzu können beispielsweise C++ *Templates* verwendet werden. Jede Ebene wird dann, wie in Abschnitt 2.2.2 bereits dargestellt wurde, in Form eines *Mixins* implementiert. Eine Instanziierung eines solchen könnte dann durch den Anwendungscode wie folgt geschehen:

```
typedef FAT<Harddisk> Diskfilesystem;
```

Daneben könnte die Anwendung sogar weitere Ausprägungen generieren lassen:

```
typedef ISO9660<CDROM> CDFilesystem;
typedef FAT<Floppy> FloppyFilesystem;
```

Sofern zum Konfigurierungszeitpunkt der Quellcode zur Verfügung stehen kann, lässt sich das Problem der kombinatorischen “Explosion” durch diesen Ansatz lösen. Er weist aber auch einen Nachteil auf:

- *Template*-basierte Codegenerierung birgt grundsätzlich das Risiko der Codeduplikation. Dies gilt umso stärker, je mehr Ebenen das System besitzt. Die Skalierbarkeit des Ansatzes ist also fraglich.

Dieser Nachteil kann vermieden werden, wenn statt der starren statischen Bindung, wie sie in den bis hier diskutierten Ansätzen verfolgt wurde, auf dynamisches Binden gesetzt wird. In Abschnitt 2.1 wurde bereits gezeigt, dass dies mit objektorientierter Software elegant sprachlich umgesetzt werden kann. Abbildung 4.5 zeigt ein entsprechendes Klassendiagramm. Der Entwurf beruht auf dem Entwurfsmuster namens *Strategy* [46]. Wie in dem Diagramm zu erkennen ist, benutzen hier die Klassen der Dateisystemebene die Gerätetreiber ausschließlich über eine Schnittstellenklasse. Die übliche Implementierung dieser Struktur in C++ basiert auf den sogenannten rein virtuellen Funktionen. Durch diese Technik wird erreicht, dass ein Funktionsaufruf zur Laufzeit an eine vom jeweiligen Zielobjekttyp abhängige Funktionsimplementierung geleitet wird. Der Maschinencode einer aufrufenden Funktion ist dabei unabhängig von der Zielfunktion womit die Notwendigkeit, mehrere Varianten davon zu generieren, entfällt. Es kommt daher nicht zu Codeduplikationen.

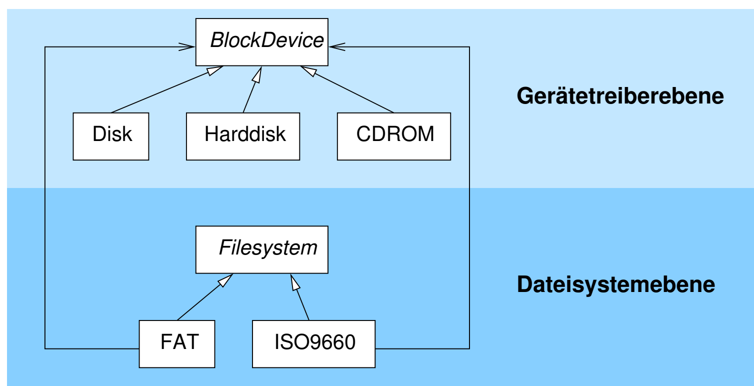


Abbildung 4.5: Dynamisches Binden von Dateisystemimplementierungen an verschiedene Blocktreiber

Trotz dieser Vorteile gegenüber den bisher vorgestellten Implementierungsstrukturen gibt es auch einen Nachteil:

- Dynamisches Binden kostet zur Laufzeit sowohl Speicherplatz als auch Rechenzeit. Dabei kommt dieser Ressourcenverbrauch nicht nur durch den indirekten Sprung und die dafür benötigten Datenstrukturen zustande, sondern

vor allem dadurch, dass so aufgerufene Funktionen nicht per *Inlining* an der Aufrufstelle eingebettet werdet können.

Die Auswahl der einen, "richtigen" Implementierungsstruktur ist für einen Entwickler sehr schwierig. Während die statische Entscheidung für jeweils eine Variante einer Ebene, wie sie bisher bei PURE vielfach durchgeführt wurde, den effizientesten Code verspricht, bieten sowohl die *Template* Variante als auch die Variante, die auf dynamischem Binden basiert, ein höheres Maß an Wiederverwendbarkeit. Hier zeigt sich der in Abschnitt 4.1.1 beschriebene Konflikt zwischen Effizienz und Wiederverwendbarkeit. Welche der wiederverwendbaren Varianten den besseren Code liefert, hängt von den jeweiligen Subsystemen, aber auch vom Anwendungsprofil ab. Bei der Verbindung relativ großer Schichtimplementierungen wie Dateisystemen und Gerätetreibern werden die zusätzlichen Kosten für dynamisches Binden weniger ins Gewicht fallen als bei kleinen Systemkomponenten wie Listenverwaltern und ihren Sortierstrategien. Das Risiko der Codeduplikation bei der *Template*-basierten Codegenerierung ist dagegen bei großen Subsystemen besonders schwerwiegend. Damit besteht eine sinnvolle Herangehensweise darin, große Subsysteme dynamisch und Klassen im Kleinen statisch zu verbinden. Problematisch ist daran, die richtige Grenze zu finden. Diese Grenze hängt vom Übersetzer, der Konfigurierung der zu verbindenen Schichtimplementierungen und dem Anwendungsprofil ab. All dies wäre gar kein Problem, wenn die Implementierungsstruktur leicht geändert werden könnte, doch leider handelt es sich auch hier um ein *Crosscutting Concern*.

Obwohl die bisherigen Erläuterungen sich in den Beispielen auf C++ als Implementierungssprache bezogen, ist dieses Problem von der Programmiersprache unabhängig. Die Wahl zwischen der Beschränkung auf eine Codevariante, der Generierung mehrerer Varianten oder einer flexiblen Variante besteht grundsätzlich.

4.3 Lösungsansatz

Die Behandlung von *Crosscutting Concerns* und die "richtige" Implementierungsstruktur zu finden, waren bei der Entwicklung von PURE neben dem administrativen Problem mögliche Konfigurationen zu verwalten, die Haupthindernisse. Die aspektorientierte Programmierung (AOP) verspricht hier Abhilfe. So könnten im Sinne des Prinzips *Separation of Concerns* die einzelnen Systembausteine vom Code zur Implementierung eines *Crosscutting Concerns* befreit werden. Auf diese Weise erhält man sehr lose gekoppelte und wiederverwendbare Systembausteine.

Auch die Implementierungsstruktur kann als *Crosscutting Concern* betrachtet werden. Letztlich realisiert sie nur eine nicht-funktionale Systemeigenschaft: den Ressourcenverbrauch. Wenn man die Struktur eines Subsystems separat durch ein Aspektprogramm beschreiben könnte, wäre es möglich, diese in Abhängigkeit vom Anwendungsszenario zu konfigurieren und damit die Effizienz der Implementierung zu verbessern. Gleichzeitig würde der Entwurfsvorgang vereinfacht werden, da der Entwickler sich nicht langfristig auf eine Struktur festlegen muss. Dies wäre

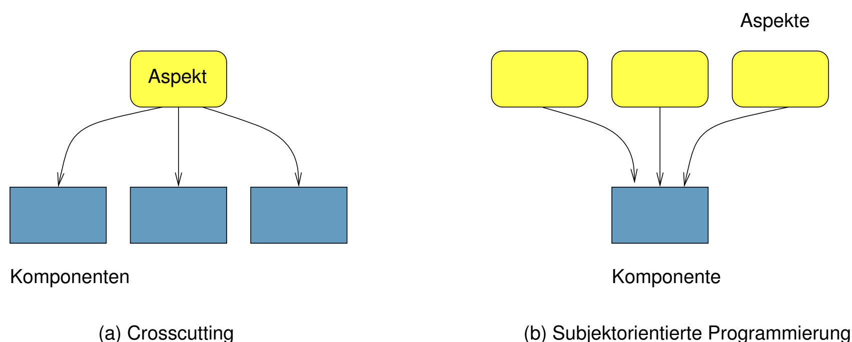
ganz im Sinne des Familienansatzes, bei dem angestrebt wird, Entwurfsentscheidungen, die die Vergrößerung der Familie behindern könnten, so lange wie möglich hinauszuzögern. Ansätze für eine solche Trennung von Struktur und Algorithmen wurden in den vergangenen Jahren bereits verfolgt [73].

Die folgenden Abschnitte beschäftigen sich mit den Möglichkeiten, die der Einsatz aspektorientierter Programmierung in Programmfamilien erschließt. Die notwendigen Modellerweiterungen zur Unterstützung des Entwurfsprozesses und die Werkzeuge, die für die Implementierung benötigt werden, sind Gegenstand weiterer Kapitel.

4.3.1 Aspekte im Einzelsystem

Das typische Anwendungsszenario für Aspekte als programmiersprachliches Konstrukt ist die Implementierung eines *Crosscutting Concerns* durch das Einwirken auf verschiedene Systemkomponenten (siehe Abbildung 4.6 (a)). Als Beispiel soll hier Überwachungscode dienen. Viele Betriebssysteme erlauben die Überwachung von Systemzuständen und Laufzeiten für bestimmte Operationen. Da die Informationen über die interessanten Systemzustände nicht notwendigerweise an einer Stelle gebündelt vorliegen, muss der entsprechende Überwachungscode, der die Information aus dem System herausreicht, verstreut vorliegen. Dies gilt insbesondere auch für den Code, der Laufzeiten ermittelt. Sollen zum Beispiel Durchlaufzeiten durch den Systemkern bestimmt werden, müssen alle möglichen Kerneintrittspunkte Code enthalten, der Zeitstempel aufzeichnen oder im laufenden Betrieb wegschicken kann.

Eine weitere Form der aspektorientierten Programmierung ist die von Hyper/J oder FOG unterstützte subjektorientierte Programmierung. Dabei wird zusätzlich zum normalen Einwirken eines Aspekts auf verschiedene Komponenten vorgesehen, dass einzelne Klassen oder Objekte aufgrund mehrerer Aspekte aus Fragmenten zusammengesetzt werden (siehe Abbildung 4.6 (b)). Der Vorteil dieses Ansatzes ist, dass Daten und Operationen, die nur aus bestimmten Blickwinkeln des Systems auf das jeweilige Objekt interessant sind, sauber getrennt werden können.



(a) Crosscutting

(b) Subjektorientierte Programmierung

Abbildung 4.6: Übliche Einsatzformen für Aspekte

Ein Beispiel aus dem Betriebssystemsektor, bei dem sich die subjektorientierte Programmierung anbieten würde, wäre der sogenannte Prozesskontrollblock mit seinen Manipulationsfunktionen. Diese Datenstruktur beinhaltet in klassischen Systemen alle für das Betriebssystem relevanten Informationen über einen Prozess. Je nach Perspektive sind dies jedoch unterschiedliche Informationen. So interessiert die Speicherverwaltung beispielsweise, welche Speicherbereiche dem Prozess zugeordnet sind, aber nicht, welcher Prozess als nächstes in der Prozesswarteschlange des Prozess-*Schedulers* anzutreffen ist. Durch subjektorientierte Programmierung könnten diese Informationen im Quelltext auseinandergehalten werden, der Zugriffsschutzmechanismus der Programmiersprache könnte dafür sorgen, dass Speicherverwaltung und Scheduler nicht auf die Daten des jeweils anderen zugreifen können und trotzdem könnten alle Daten im laufenden System platzsparend gemeinsam abgelegt werden.

4.3.2 Aspekte in der Programmfamilie

Beide im letzten Abschnitt aufgeführten Anwendungsfälle treten sowohl in gewöhnlichen Softwaresystemen als auch in Programmfamilien auf. Darüber hinaus entstehen im Kontext von Programmfamilien durch Aspekte interessante neue Möglichkeiten.

Der erste Fall besteht in der Konfigurierung des Aspektcodes, das heißt, es wirken unterschiedliche Varianten eines Aspekts auf dasselbe Programm. Dadurch kann ein Aspekt je nach Konfigurierung beispielsweise auf völlig unterschiedliche Punkte oder auch gar nicht im System wirken. Im Beispiel der Systemüberwachung ist dies speziell vor dem Hintergrund des Sparens von Ressourcen ein großer Vorteil. So kann entsprechend der Messanforderungen eine individuelle Instrumentierung mit Überwachungscode erfolgen, wodurch das unnötige Aufzeichnen von Informationen vermieden wird. Einige Arbeiten haben bereits gezeigt, dass so sehr schlanke Überwachungssysteme erstellt werden können, die das vermessene System nur minimal belasten [76].

Der zweite interessante Fall entsteht dadurch, dass in einer Programmfamilie natürlich auch der Komponentencode konfiguriert wird. So kann zum Beispiel ein Aspekt auf unterschiedliche Varianten des gleichen Subsystems wirken und so bestimmten Code ausfaktorisieren, der ohne Aspekte in jeder Variante repliziert werden müsste. Als Beispiel sei hier eine Familie trigonometrischer Funktionen genannt. Die Familie soll verschiedene Varianten der Berechnung bieten, von denen je nach Anforderungen an Genauigkeit und Ressourcenverbrauch eine ausgewählt wird. Völlig unabhängig davon soll nun konfiguriert werden können, ob die jeweilige Funktion die Werte der Eingangsparameter auf Sinnhaftigkeit überprüfen soll. Ohne AOP würde dies bedeuten, in jeder Variante den Überprüfungscode – oder mindestens einen Aufruf – einbauen zu müssen, der durch bedingte Übersetzung deaktivierbar sein muss. Ein Aspekt dagegen kann den Überprüfungscode von den trigonometrischen Funktionen trennen und ihn zur Übersetzungszeit zur ausgewählten Variante hinzufügen.

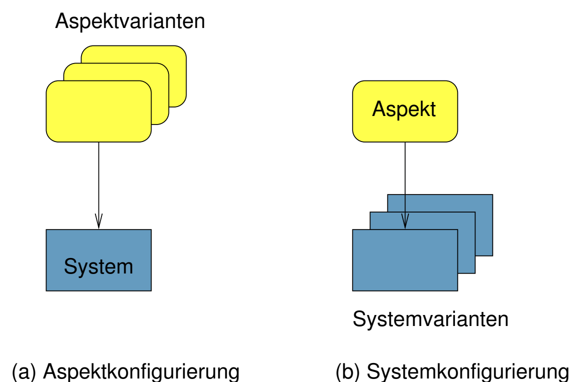


Abbildung 4.7: Aspekte und Konfigurierung

Abbildung 4.7 veranschaulicht die beiden genannten Fälle zusammenfassend noch einmal. Die hier gennanten Beispiele sind klein und überschaubar. Typische reale Aspekte in Betriebssystemcode wie der Aspekt zur Unterbrechungssynchronisation, der in Kapitel 9 vorgestellt werden wird, umfassen das normale *Crosscutting* eines Aspekts über mehrere Module, sind konfigurierbar und wirken auf unterschiedliche Systemvarianten. Das heißt, die Kombination der hier beschriebenen Fälle ist die Regel.

4.3.3 Zu erwartende Vorteile

Durch die modulare Implementierung von *Crosscutting Concerns* mit Hilfe von AOP wird hauptsächlich erreicht, dass Änderungen an dieser Implementierung wesentlich einfacher und damit risikoloser sind, als ohne die Modularisierung. Indirekt wird damit auch das Erstellen und Warten von Programmfamilien erleichtert, da durch AOP ein einziger Konfigurationsschalter im Aspektcode große Auswirkungen auf das System haben kann. Im Vergleich zu herkömmlichen Implementierungen wird somit eine Konzentration vieler Konfigurationsentscheidungen in einem einzigen Konfigurationspunkt erreicht. Betrachtet man eine gesamte Programmfamilie, lässt sich so die Anzahl der Konfigurationspunkte drastisch reduzieren. Abbildung 4.8 auf der nächsten Seite soll diese Eigenschaft nochmals verdeutlichen. Sie zeigt einen Konfigurationsschalter, der eine von drei Aspektcodevarianten selektieren kann. Durch den Aspektweber werden die Systemkomponenten mit dem ausgewählten Aspektcode verbunden. In diesem Fall wird aufgrund des Aspekts an fünf Stellen manipuliert. Das heißt, aus fünf Konfigurationspunkten, wie man sie in einer herkömmlichen Implementierung gehabt hätte, wurde hier durch den Einsatz von AOP ein einziger. In vielen Fällen ist die Reduktion noch wesentlich größer.

Crosscutting Concerns korrespondieren in der Regel mit **strategischen Entwurfsentscheidungen**. Dabei bedeutet “strategisch”, dass diese Entscheidungen

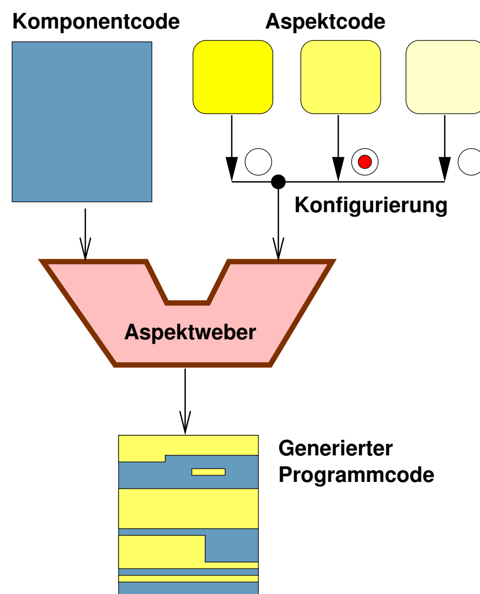


Abbildung 4.8: Reduktion von Konfigurationspunkten durch konfigurierbare Aspekte

nicht leicht wieder rückgängig zu machen sind. Durch den Einsatz von AOP könnte sich dies ändern. *Crosscutting Concerns* lassen sich modular implementieren und der Anpassungsaufwand aufgrund modifizierter Entwurfsentscheidungen hält sich in Grenzen. Damit verlieren viele strategische Entwurfseigenschaften von Betriebssystemen ihren strategischen Charakter. Im Rahmen einer Betriebssystemfamilie können die entsprechenden Eigenschaften nun sogar konfiguriert werden.

4.4 Zusammenfassung

In diesem Kapitel wurde zunächst als Hintergrundinformation erörtert, wie durch Konfigurierungsmaßnahmen der Konflikt zwischen Wiederverwendbarkeit und Effizienz in den Griff zu bekommen ist. Solche Maßnahmen können sowohl zur Übersetzungszeit als auch zur Laufzeit durchgeführt werden. Im ersten Fall handelt es sich um die klassische Programmfamilie, der zweite Ansatz führt zu dynamisch rekonfigurierbaren Softwaresystemen. In dieser Arbeit liegt der Schwerpunkt auf den statisch konfigurierbaren Systemen. Mit ihrer Hilfe kann man auch für kleinste eingebettete Systeme ressourcensparende Betriebssysteme bauen, deren Grundlage wiederverwendbare Systembausteine sind. Anhand der PURE Betriebssystemfamilie wurde gezeigt, wie diese Idee in den letzten Jahren mit Hilfe eines auf einer funktionalen Hierarchie basierenden Entwurfs und einer objektorientierten Implementierung umgesetzt wurde. Die Ergebnisse dieser Entwicklung bestätigen den Wert des Ansatzes.

Das Besondere an der Handhabung einer Programmfamilie ist die Beherrschung der Variantenvielfalt, die durch eine große Zahl von Konfigurierungsmöglichkeiten und ihre Abhängigkeiten entsteht. Durch die angestrebte feingranulare Konfigurierbarkeit stoßen Entwickler hier schnell an die Grenzen der Überschaubarkeit. Dieses Problem lässt sich aus zwei Richtungen angehen. Zum einen kann man Methoden entwickeln, um die Konfigurationsvielfalt durch Werkzeugunterstützung besser beherrschbar zu machen. Da hierzu parallel Arbeiten durchgeführt werden, wird dieser Ansatz in der vorliegenden Arbeit ausgeklammert. Zum anderen kann man versuchen, die Zahl der Konfigurierungspunkte auf wenige zu bündeln, wenn viele Konfigurationspunkte im System dazu dienen, gemeinsam eine bestimmte Systemeigenschaft zu implementieren. Dadurch ändern sich nicht die nach außen sichtbaren Systemeigenschaften und deren Variablität, wie sie zum Beispiel durch ein *Feature Modell* repräsentiert werden. Dagegen kann sich die Les- und Wartbarkeit der Quelltexte erheblich verbessern.

Ein programmiersprachlicher Ansatz, der dies ermöglicht, ist die aspektorientierte Programmierung. Sie erlaubt die modulare Implementierung sogenannter *Crosscutting Concerns* durch Aspekte. Im Vergleich zu Einzelsystemen ergeben sich durch AOP in der Programmfamilie noch sehr viel interessantere Anwendungen. So kann sowohl das System, auf das ein Aspekt einwirkt, als auch der Aspekt selbst konfiguriert werden. Eine kleine Änderung kann damit eine große Wirkung zeigen. So besteht die Hoffnung, dass die strategischen Entwurfsentscheidungen beim Betriebssystembau, wie zum Beispiel Schutz-, Synchronisations- oder *Caching*-Strategien, ihren "Schrecken" verlieren.

Vor der Verwirklichung dieser Idee in einer konkreten Betriebssystemfamilie stehen jedoch große Hindernisse. So müssen zunächst die Entwurfsmodelle für Programmfamilien mit dem Konzept des Aspekts in Einklang gebracht werden und die für die Umsetzung notwendigen Werkzeuge bzw. Sprachen sind zu konzipieren und zu implementieren.

In den folgenden Kapiteln wird es darum gehen, diese Hindernisse aus dem Weg zu räumen und zu zeigen, dass die verprochenen Vorteile des Ansatzes, das Familienkonzept mit aspektorientierter Programmierung im Betriebssystembau zu verbinden, tatsächlich eintreten. Dazu werden im nächsten Kapitel zunächst die konzeptionellen Grundlagen gelegt. Gefolgt wird dies von mehreren Kapiteln, die den Entwurf und die Implementierung notwendiger Werkzeuge dokumentieren und einem Kapitel mit Fallstudien, bei denen AOP im Kontext der Betriebssystemfamilie PURE angewendet wird.

Kapitel 5

AOP in Programmfamilien

Gegenstand dieses Kapitels sind die notwendigen Modelle und Werkzeuge für den aspektorientierten Entwurf von Programmfamilien und die implementierungstechnische Umsetzung. Damit werden die Grundlagen für Fallstudien geschaffen, mit deren Hilfe untersucht werden soll, ob die in Abschnitt 4.3.3 dargestellten Vorteile des Ansatzes, AOP mit dem Familienkonzept im Betriebssystembau zu kombinieren, tatsächlich eintreten.

5.1 Unterstützung des Entwurfsprozesses

Der Entwurf von Software ist eine hochkomplexe Aufgabe. Deshalb verwenden Entwickler abstrakte Modelle, die Details verbergen. Schrittweise wird nach einer Analyse ein zunächst grober Entwurf um immer mehr Informationen angereichert, bis eine erste Implementierung sinnvoll möglich ist. Dieser Vorgang kann sich vielfach wiederholen, wenn beispielsweise nach dem Spiralmodell vorgegangen wird [20]. Besonders Analyse- und Entwurfsmodelle, die eine grafische Repräsentation erlauben, sind wegen ihrer Übersichtlichkeit hilfreich.

Die folgenden Abschnitte diskutieren einige solcher Modelle unter dem Gesichtspunkt einer geplanten aspektorientierten Implementierung einer Programmfamilie. Anschließend werden die Beziehungen, die Aspekte mit Komponentencode und anderen Aspekten eingehen, untersucht, um dann, wo es notwendig erscheint, Erweiterungen der Modelle vorzunehmen.

5.1.1 Feature Modelle

Bereits im letzten Kapitel wurde angesprochen, dass Feature Modelle in Hinblick auf AOP keinerlei Ergänzung bedürfen. Wenn sie zur Entwicklung einer Programmfamilie eingesetzt werden, beschreiben sie die möglichen Eigenschaften der Familie aus Sicht eines Anwendungsentwicklers, der das Betriebssystem konfigurieren können soll. Ob diese Eigenschaften *Crosscutting Concerns* sind, ist zu dem frühen Zeitpunkt, zu dem Feature Modelle angefertigt werden, noch nicht relevant.

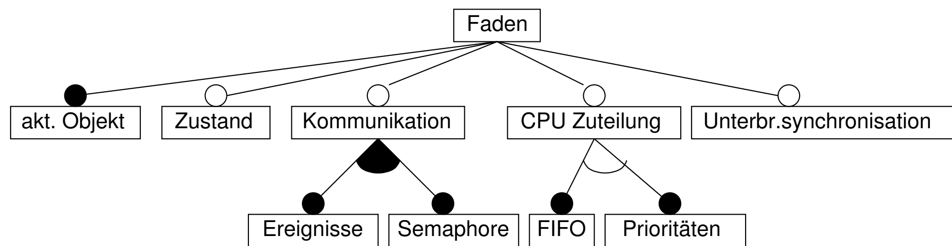


Abbildung 5.1: Feature Diagramm des Konzepts Kontrollfaden

Abbildung 5.1 zeigt beispielhaft ein Feature Diagramm, den Hauptbestandteil eines Feature Modells, für das Konzept des Kontrollfadens in einem Betriebssystem. Es zeigt notwendige und optionale Systemeigenschaften, die aus Sicht der Anwendung relevant sind, und setzt diese in Beziehung. Ob zum Beispiel die Unterbrechungssynchronisation ein *Crosscuting Concern* ist, interessiert im Feature Modell noch nicht. Hier wird nur beschrieben, dass Kontrollfäden optional über die Eigenschaft Unterbrechungssynchronisation zu betreiben verfügen. Das Beispiel des Fadenkonzepts wird in den folgenden Abschnitten nochmals aufgegriffen.

Wenn wie bei der Entwicklung von PURE die Auswahl einer Featuremenge als Ausgangspunkt für den Konfigurierungsvorgang benutzt wird und ein bestimmtes Feature ein *Crosscutting Concern* ist, ergänzen sich Aspekte und Feature Modelle sogar ideal. In diesem Fall ergibt sich nämlich eine direkte Abbildung von dem selektierten Feature auf den Aspekt. Bei einer nicht aspektorientierten Implementierung wäre die Abbildung viel komplexer, da zahlreiche Konfigurierungsschalter von dem Feature abhängen würden.

5.1.2 UML Diagramme

Die *Unified Modeling Language* (UML) ist eine inzwischen weit verbreitete und von der *Object Management Group* (OMG) zum Standard erklärte Sprache zur Modellierung von Softwaresystem [87]. Die Modelle können in Form von UML Diagrammen grafisch repräsentiert werden. Bekannte Beispiele sind die in UML integrierten Klassen- und Zustandsdiagramme.

Da UML über ein erweiterbares Metamodell verfügt, mit dem sich die Sprache selbst beschreibt, gibt es wohl kaum Zusammenhänge, die sich nicht mit UML modellieren lassen. Dies setzt jedoch eine entsprechende Erweiterung voraus. Die exisitierenden UML Diagrammformen sind nicht speziell für die Modellierung von aspektorientierten Softwaresystemen oder den Beziehungen zwischen Schichten einer Programmfamilie gedacht und daher wenig geeignet.

Vorschläge für UML Erweiterungen, die Aspekte berücksichtigen, wurden bereits mehrfach publiziert [3, 23, 112]. Alle diese Vorschläge haben allerdings gemein, dass sie Aspekte ähnlich wie Klassen behandeln. Dies ist ideal, wenn man an

aspektorientierte Spracherweiterungen objektorientierter Sprachen denkt. Im Kontext von Programmfamilien ist ein direkter Schritt von einem Feature Modell zu einem Klassendiagramm jedoch zu groß. Hier bräuchte man noch weitere Diagramme in der Art von funktionalen Hierarchien oder GenVoca Modellen, die Funktionen in Beziehung setzen bzw. konfigurierbare Subsysteme modellieren. Nur darüber findet ein Entwickler eine Modulstruktur, die sich statisch konfigurieren lässt, und so der Variabilität, die durch das Feature Modell beschrieben wird, gerecht wird. Prinzipiell ist es denkbar, solche Modelle durch Erweiterungen in UML zu integrieren. Viel wichtiger ist es allerdings, vorher herauszufinden, wie GenVoca Modelle und funktionale Hierarchien in Hinblick auf AOP überhaupt gestaltet sein müssen, um gegebenenfalls Erweiterungen einzuführen.

5.1.3 GenVoca Modell versus funktionale Hierarchie

Funktionale Hierarchien und GenVoca Modelle sind für die frühen Phasen des Entwurfs einer Programmfamilie sehr gut geeignet. Wegen ihrer Einfachheit und Nähe zur Modulstruktur sind sie eine ideale Ergänzung zu den rein eigenschaftsorientierten Feature Modellen. Allerdings fokussieren sie auf funktionale Eigenschaften bzw. Systemkomponenten. Nicht-funktionale Eigenschaften, also das “wie” der Implementierung, werden, trotz ihrer Wichtigkeit gerade im Betriebssystemsektor, vernachlässigt und auch die andersartigen Beziehungen zwischen Aspekten und Komponentencode sind auf den ersten Blick nicht mit diesen Modellen zu beschreiben. Es ist daher wünschenswert ein erweitertes Modell zu entwickeln, bei dem nicht-funktionale Eigenschaften und Aspekte, die ganz allgemein zur Implementierung von *Crosscutting Concerns* eingesetzt werden können, berücksichtigt werden.

Abbildung 5.2 auf der nächsten Seite legt nahe, GenVoca Modelle als Verfeinerung funktionaler Hierarchien zu betrachten. Links zeigt sie eine funktionale Hierarchie, die die notwendigen Funktionen umfasst, um die konfigurierbaren Eigenschaften des Kontrollfadenkonzepts aus Abbildung 5.1 zu erbringen. Sie wurde der funktionalen Hierarchie der PURE Fadenverwaltung nachempfunden. Rechts ist eine als Verfeinerung anzusehende GenVoca Grammatik zu erkennen, bei der Systemkomponenten und konfigurierbare Schichten entsprechend der Schichten der funktionalen Hierarchie angeordnet sind. Damit erfolgt bei diesem Entwurfsschritt ein Übergang von abstrakten Funktionen zu einer konkreten Modulstruktur. Der nächste Schritt könnte beispielsweise darin bestehen, die Komponenten in Form von Klassen oder Gruppen von Klassen auszudrücken und die Schnittstellen der Schichten festzulegen. Das Ergebnis dieses Vorgangs könnte wie die entsprechende Klassenhierachie von PURE in Abbildung 4.3 auf Seite 38 aussehen.

Die zusätzlichen Informationen im GenVoca Modell umfassen zum einen die Konfigurierbarkeit einzelner Ebenen, was zum Beispiel bei der PURE Familie genutzt wurde. Zum anderen können nicht aspektorientiert implementierte nicht-funktionale Eigenschaften mit Hilfe von Attributen sinnvoll für die Systemkonfigurierung genutzt werden (siehe Abschnitt 2.2.2). Zudem können anhand der Gen-

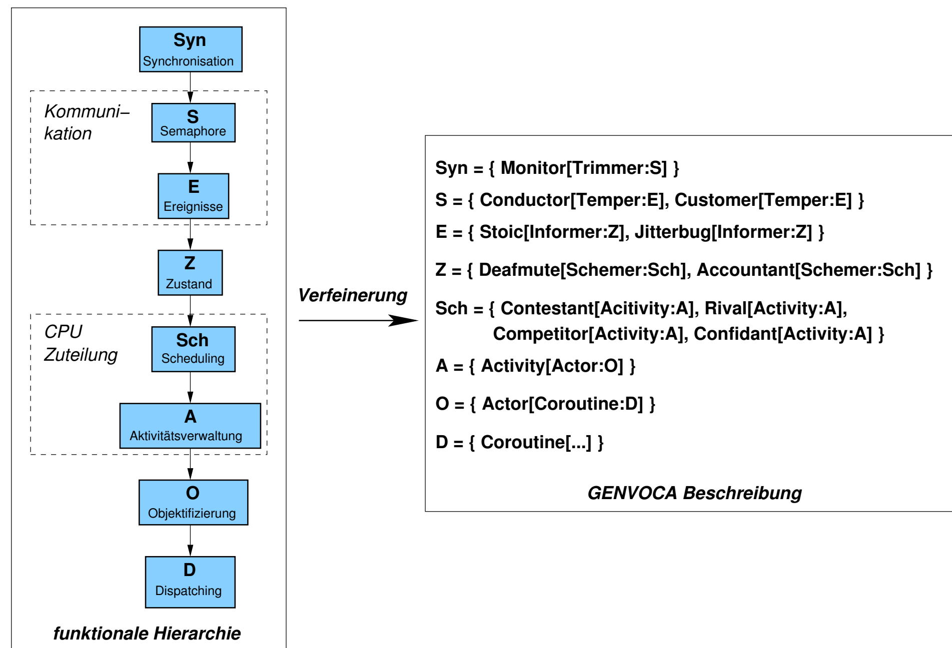


Abbildung 5.2: Verfeinerung einer funktionalen Hierarchie mit Hilfe einer GenVoca Beschreibung

Voca Grammatik richtige aber ansonsten sinnlose Konfigurationen mit Hilfe der sogenannten *Design Rules* erkannt werden. All dies geht über die Modellierungsfähigkeiten funktionaler Hierarchien hinaus.

Betrachtet man, wie hier anhand des Beispiels gezeigt wurde, ein GenVoca Modell als Verfeinerung einer funktionalen Hierarchie, würde daraus resultieren, dass die Modelle zu unterschiedlichen Zeitpunkten im Entwurfsprozess eingesetzt würden. Damit könnte auf keines verzichtet werden und beide Modelle müssten in Hinsicht auf AOP erweitert werden.

Leider ist jedoch der Übergang von der funktionalen Hierarchie zum GenVoca Modell in den meisten Fällen nicht so leicht wie in diesem Beispiel. Der Grund besteht darin, dass eine funktionale Hierarchie ein ganz allgemeiner gerichteter azyklischer Graph ist, während sich beim GenVoca Modell durch Ableitung der Grammatik lediglich baumartige Systemstrukturen konstruieren lassen. Um von einer allgemeinen funktionalen Hierarchie zu einem GenVoca Modell zu kommen, müsste man daher in einigen Fällen an sich unabhängige Funktionen zu einer gemeinsamen Schicht zusammenfassen oder sie in getrennte Schichten legen, womit jedoch unnötigerweise eine Ordnung zwischen ihnen festzulegen wäre.

Um solche Probleme zu vermeiden, wird im Folgenden für eine aspektorientierte Erweiterung auf die Verwendung des GenVoca Modells verzichtet. Dafür wird die Idee der konfigurierbaren Schichten aufgegriffen und als Erweiterung der funktionalen Hierarchien in Form konfigurierbarer Funktionen aufgenommen.

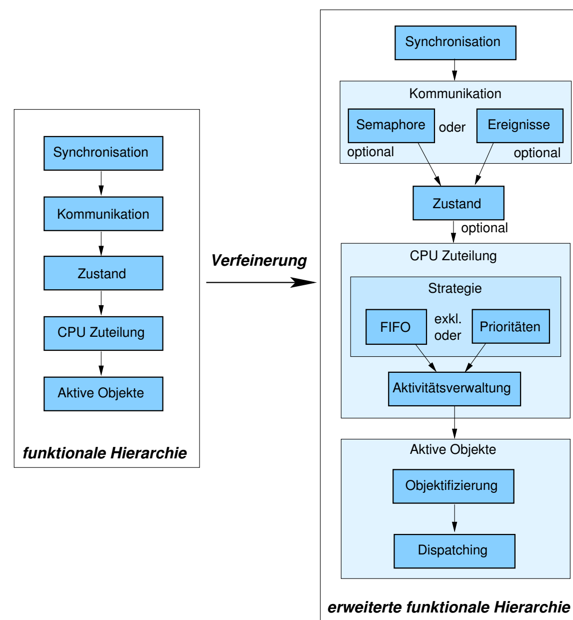


Abbildung 5.3: Verfeinerung einer funktionalen Hierarchie mittels konfigurierbarer Funktionen

Abbildung 5.3 zeigt, wie auf diese Weise funktionale Hierarchien schrittweise verfeinert und konfigurierbar gestaltet werden können. Beispielsweise wurde hier die Funktion "Aktive Objekte" verfeinert. Um aktive Objekte bereitzustellen wird eine Funktion "Objektifizierung" benötigt, die auf einer Funktion "Dispatching" basiert. Konfigurierbar sind in der Abbildung die Funktionen "Strategie" und "Kommunikation". Der Entscheidung für eine konfigurierbare Funktion "Strategie" liegt beispielsweise der Gedanke zugrunde, dass es für die CPU Zuteilung zahlreiche Strategien gibt, von denen jeweils nur eine eingesetzt werden soll. Alle anderen Funktionen sollen von dieser Entscheidung unabhängig sein. Dies ist natürlich nur möglich, wenn die späteren Implementierungen der Funktionsvarianten kompatible Schnittstellen aufweisen. Ist diese Voraussetzung gegeben, entsteht durch eine konfigurierbare Funktion eine Art "Familie in der Familie", wodurch die hier beschriebene Entwurfsmethodik rekursiv angewendet werden kann. CPU Zuteilungsstrategien können so völlig getrennt als eigene Familie modelliert und weiter verfeinert werden. Der ganze Prozess endet, wenn die Granularität der Funktionen so klein ist, dass eine direkte Abbildung auf Modularisierungskonstrukte der Programmiersprache wie Klassen sinnvoll ist.

5.1.4 Beziehungen von Aspekten zu Komponentencode

Die Aufgabe der funktionalen Hierarchien besteht in der Beschreibung der Beziehungen von Funktionen. Um *Crosscutting Concerns* bzw. Aspekte in dieses Ent-

wurfsmodell integrieren zu können, müssen zunächst deren Beziehungen beleuchtet werden. Daher werden in diesem Abschnitt die Wechselwirkungen von Aspekten und Komponentencode und im nächsten Abschnitt von Aspekten und anderen Aspekten betrachtet. Der Begriff "Komponente" bezieht sich an dieser Stelle auf das in Abschnitt 2.3 eingeführte, im AOP Bereich übliche Vokabular und setzt nicht notwendigerweise eine komponentenorientierte Entwicklung voraus [103].

Ein erstes formales Modell von AOP [36] beschreibt einen Aspektweber als einen "generischen Systemmonitor", der Zustände überwacht und gegebenfalls Instruktionen einfügt. Ein Aspekt beschreibt diese Zustände und Instruktionen[1]. Damit wird ähnlich wie bei der Behandlung einer Ausnahme durch den Prozessor, zum Beispiel bei einer Division durch 0, Aspektcode aus dem normalen Kontrollfluss heraus aktiviert, wenn die passende Bedingung eintritt. Dazu ist ein ausdrücklicher Aufruf nicht nötig. Dieser Vorgang wird im Folgenden als **Aktivierung** von Aspektcode bezeichnet.

Gleichzeitig existiert noch eine zweite, entgegengesetzt gerichtete Beziehung. So basieren die Beschreibungen der Systemzustände typischerweise auf Namen von Attributen oder Funktionen. Ein sehr gebräuchlicher Ausdruck zur Beschreibung eines solchen Systemzustands ist zum Beispiel "das Erreichen eines Aufrufs der Funktion X". Die Komponenten, auf die ein Aspekt einwirken soll, müssen daher zumindest bekannt sein. In manchen Fällen ist es sogar notwendig, dass die beschriebenen Systemzustände und daraus resultierenden Verbindungspunkte tatsächlich vorhanden sind, damit der Aspekt wie erwartet arbeitet. Somit führt auch das **Einwirken** eines Aspekts auf andere Komponenten des Systems zu einer relevanten Beziehung.

Da die Aktivierung eine Folge des Einwirkens ist und Aktivierungen ohne Einwirkung nicht stattfinden, kann beides zusammen als eine einzige bidirektionale Beziehung zwischen Aspekt- und Komponentencode angesehen werden.

Mit dieser Beziehung ist nicht notwendigerweise eine Abhängigkeit verbunden. So können Aspekte existieren, für die es völlig unproblematisch ist, wenn die von ihnen beschriebenen Systemzustände nie eintreten und dementsprechend der Aspektcode nicht aktiviert wird. Gleichzeitig "funktionieren" auch viele Systemkomponenten unabhängig davon, ob ein bestimmter Aspekt auf sie einwirkt. Am leichtesten wird dies am Beispiel eines Aspekts deutlich, der zur Fehlersuche eingesetzt wird. Der Aspekt beschreibt, dass beim Erreichen eines Aufrufs einer bestimmten Funktion eine Ausgabe erfolgen soll. Es ist aus Sicht des Aspekts kein Problem und er hat seine Aufgabe vollständig erfüllt, auch wenn es nie zu einem solchen Aufruf kommt. Gleichzeitig ist das korrekte Verhalten des Komponentencodes nicht von dem Einwirken dieses Aspekts abhängig. Diese lose Art der Beziehung ist sehr typisch für Aspekte und macht sie für den Einsatz in Familien interessant, wo zum Beispiel das Fehlen von Komponenten in bestimmten Konfigurationen toleriert werden muss.

[1] Die einzelnen aspektorientierten Sprachen oder Werkzeuge können natürlich den Anwender im Vergleich zu diesem generischen Modell einschränken.

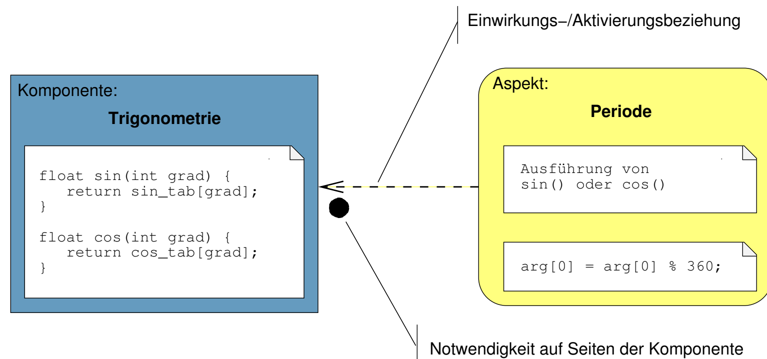


Abbildung 5.4: Notwendiges Einwirken aus Sicht des Komponentencodes

Neben dieser losen Beziehung zwischen Komponente und Aspekt ist aber auch eine engere Kopplung denkbar. Dies zeigt das Beispiel in Abbildung 5.4. Dort sorgt ein Aspekt dafür, dass die Argumente der Sinus- und Cosinusfunktion entsprechend ihrer Periodizität auf einen Wert zwischen 0 und 359 abgebildet werden. Wenn diese Einwirkung nicht stattfände, käme es in der Komponente unter Umständen zu einem Fehlverhalten, da die Implementierung auf einer Wertetabelle basiert, die nur Indexwerte zwischen 0 und 359 erlaubt. Der gestrichelte Pfeil repräsentiert die Beziehung zwischen Aspekt und Komponente. Der Punkt neben der Pfeilspitze verdeutlich, dass diese auf Seiten der Komponente notwendig ist.

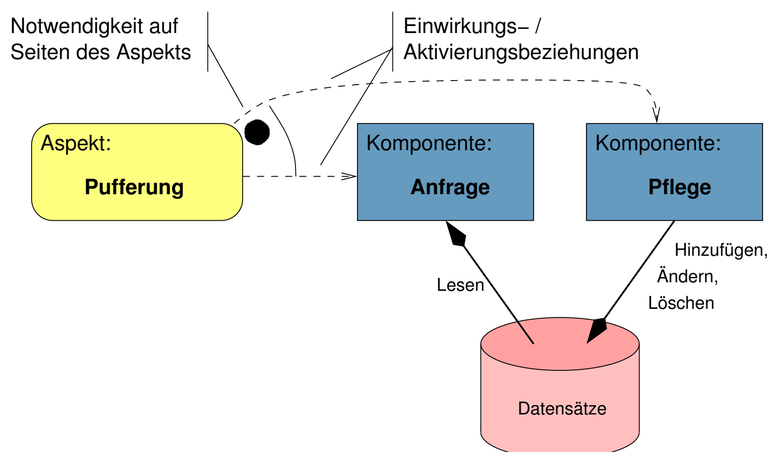


Abbildung 5.5: Notwendige Aktivierung aus Sicht des Aspekts

Auch aus Sicht eines Aspekts kann die Beziehung zu einer Komponente notwendig sein. Dies verdeutlicht das Beispiel in Abbildung 5.5. Dort puffert ein Aspekt Datensätze, die von einem langsamen Speichermedium kommen, um bei erneutem Zugriff auf dieselben Daten schneller zu sein. Die gepufferten Datensät-

ze müssen jedoch für ungültig erklärt werden, falls die Pflege-Komponente Modifikationen an den Datensätzen vornimmt. Durch ein Einwirken auf die Pflege-Komponente sorgt daher der Aspekt dafür, dass er aktiviert wird, falls es zu Modifikationen kommt. In Hinblick auf die Beziehung zur Anfrage-Komponente ist die Beziehung des Aspektes zur Pflege-Komponente damit notwendig. Dies wird durch den Punkt und den Bogen zwischen den Beziehungen symbolisiert.

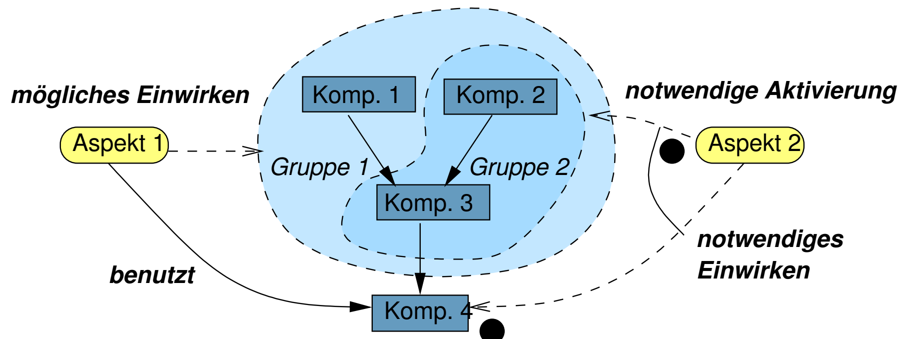


Abbildung 5.6: Beziehungen zwischen Aspekten und herkömmlichen Modulen

Eine weitere Besonderheit der Beziehung von Aspekten zu Komponentencode ist, dass davon in der Regel ganze Gruppen von Komponenten betroffen sind. Dies zeigt Abbildung 5.6. Hier wurden Kompontengruppen gebildet, auf deren Elemente ein Aspekt jeweils in gleicher Weise wirkt.

5.1.5 Beziehungen von Aspekten zu anderen Aspekten

Die Bedingung zur Aktivierung von Aspektcode kann natürlich auch innerhalb eines Aspekts wahr werden. Damit bestehen auch zwischen Aspekten Einwirkungsbeziehungen, wie sie im letzten Abschnitt beschrieben wurden[2].

Interessanter sind die indirekten Aspektinteraktionen [36]. Ein einfaches Beispiel für solche Interaktionen zeigt Abbildung 5.7 auf der nächsten Seite. Dort wirken zwei Aspekte auf denselben Verbindungspunkt der Komponente “Sender”. Der eine Aspekt soll die Kommunikation zwischen Sender und Empfänger verschlüsseln, während der zweite Aspekt die kommunizierten Daten protokolliert. Je nach Reihenfolge der Aktivierung resultiert in diesem Beispiel ein unterschiedliches Systemverhalten. In einem Fall werden die unverschlüsselten Daten protokolliert und im anderen Fall werden die Daten erst verschlüsselt und dann protokolliert. Wenn hier ein deterministisches Verhalten gewünscht ist, muss das Aspektprogramm dem

[2]Ein typischer Fehler von Programmierern, die noch wenig Erfahrung mit AOP haben, besteht darin, versehentlich *Pointcuts* so zu definieren, dass ein *Advice* auf seinen eigenen *Advice*-Code wirkt. Je nach Implementierung des Webers, kann dies zu einer Endlosrekursion zur Laufzeit oder einem nicht terminierenden Webevorgang führen. Eine automatische Erkennung dieses Problems ist je nach *Joinpoint*-Modell nicht immer einfach.

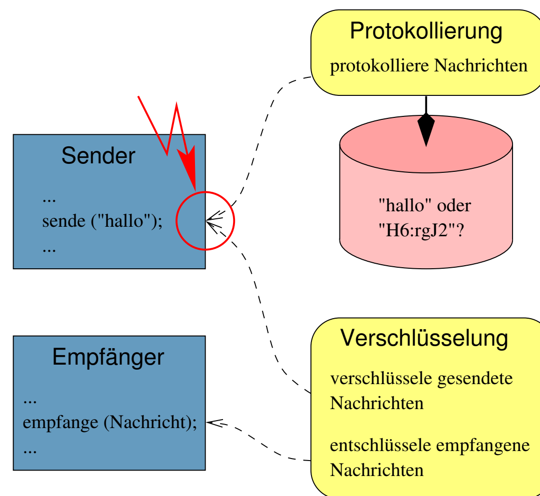


Abbildung 5.7: Aspektinteraktionen

Weber einen Hinweis geben, in welcher Reihenfolge die Aktivierungen stattzufinden haben.

AspectJ bietet zum Beispiel zu diesem Zweck das *dominates* Sprachkonstrukt:

```
aspect Id dominates TypePattern { ... }
```

Durch dieses Konstrukt wird für einen Aspekt *Id* angegeben, dass er im Falle einer Anwendung auf den gleichen Verbindungspunkt vor den Aspekten aktiviert werden soll, die durch ein Namensmuster *TypePattern* beschrieben werden[3].

Durch einen solchen Hinweis würde die Beziehung zwischen dem Protokollierungs- und dem Verschlüsselungsaspekt im Quellcode sichtbar werden. Auch bei solchen **Ordnungsbeziehungen** zwischen Aspekten gibt es den Fall, dass ein oder beide Partner von der Beziehung mit dem anderen abhängig sind. Diese wäre zum Beispiel der Fall, wenn der Verschlüsselungsaspekt in der veränderten Nachricht noch Informationen – etwa einen Code für die Verschlüsselungsmethode – unterbringen würde, die vom Protokollierungsaspekt benutzt werden. Damit wäre der Protokollierungsaspekt auf das vorherige Wirken des anderen Aspekts angewiesen. Eine solche Beziehung wird im Folgenden als **notwendige Ordnungsbeziehung** bezeichnet. Dagegen kann bei einer möglichen Ordnungsbeziehung auch jeder Aspekt ohne den anderen existieren. Die Ordnung ist nur relevant, falls beide Aspekte auf den gleichen Verbindungspunkt wirken. Abbildung 5.8 auf der nächsten Seite zeigt abschließend, wie die Aspektinteraktions- bzw. Ordungsbeziehungen fortan grafisch dargestellt werden sollen.

[3]Ein umgekehrtes Konstrukt, das gezielt anderen Aspekten den Vortritt lässt, wurde zwar bereits auf der Mailing Liste der AspectJ Nutzer diskutiert, wurde aber bisher noch nicht in die Sprache aufgenommen.

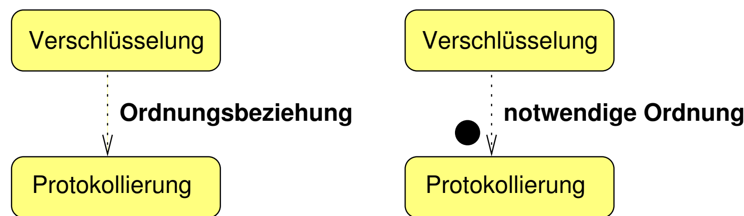


Abbildung 5.8: Ordnungsbeziehungen von Aspekten

5.1.6 *Concern*-Hierarchien

Die in den letzten Abschnitten durchgeführten Betrachtungen hatten zum Ziel, das notwendige Wissen für eine aspektorientierte Erweiterungen des Modells der funktionalen Hierarchie zu sammeln. Mit ihrer Hilfe soll bei dem Entwurf der Familie eine Modulstruktur gefunden werden, die bereits ein hohes Maß an Konfigurierbarkeit durch einfaches Weglassen bietet. Funktionen werden dabei typischerweise direkt auf Module abgebildet. Da auch Aspekte Module sind, wäre es sinnvoll, auf der Ebene der funktionalen Hierarchie *Crosscutting Concerns* mit aufzunehmen und als solche zu kennzeichnen. Beim Übergang zur Modulstruktur können diese dann analog zum Vorgehen bei Funktionen auf Aspekte abgebildet werden.

Bezüglich der Beziehungen in der Hierarchie kommt es darauf an, auf welche anderen *Crosscutting Concerns* und Funktionen das jeweilige *Crosscutting Concern* einen Einfluss hat und von welchen es abhängt. Obwohl nicht gesagt ist, dass ein *Crosscutting Concern* auch tatsächlich durch einen Aspekt modular implementiert werden kann, sind die möglichen Beziehungstypen doch mit denen identisch, die in den letzten beiden Abschnitten in Bezug auf Aspekt- und Komponentencode beschrieben wurden. Die grafische Repräsentation soll daher im Folgenden übernommen werden.

Es ist allerdings zu beachten, dass die so modellierten Beziehungen über das hinausgehen, was für die bloße Beschreibung möglicher Systemkonfigurationen erforderlich ist. “Mögliche Einwirkungsbeziehungen” können zum Beispiel unter diesem Gesichtspunkt komplett ignoriert werden, da sowohl der aus dem *Crosscutting Concern* resultierende Aspekt als auch die Module, die aus den in Beziehung stehenden Funktionen resultieren, auf die Anwesenheit des jeweils anderen verzichten können. Anders ist die Situation bei den als “notwendig” charakterisierten Beziehungen. Sie erfordern in jeder Systemkonfiguration die Anwesenheit des Partners. Dies entspricht auch den Abhängigkeitsbeziehungen in funktionalen Hierarchien. Solche Beziehungen sind in der grafischen Repräsentation leicht an dem Punkt zu erkennen. Damit ist eine Filterung der Beziehungen leicht möglich, bei der nur noch die für die Konfigurierung relevanten Abhängigkeitsbeziehungen übrig bleiben. Abbildung 5.9 auf der nächsten Seite zeigt diesen Abbildungsvorgang. Das Resultat auf der rechten Seite der Abbildung ist ein zyklenfreier gerichteter Graph, der direkt in eine Modulstruktur aus Aspekten und herkömmlichen

Modulen umgesetzt werden kann.

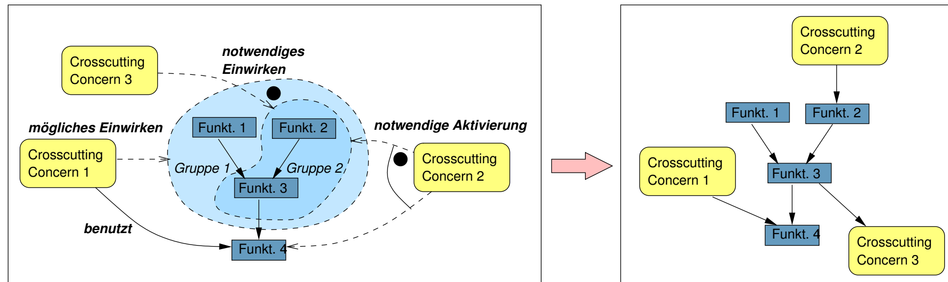


Abbildung 5.9: Abbildung von *Concern*-Beziehungen auf Abhängigkeitsbeziehungen im Sinne funktionaler Hierarchien

Bei dem vom GenVoca Modell inspirierten Konzept der austauschbaren Funktionen ist die Austauschbarkeit genau dann gegeben, wenn die aus den Funktionen resultierenden Module eine kompatible Schnittstelle ausweisen. Da mit erweiterten funktionalen Hierarchien auch *Crosscutting Concerns* modelliert werden und diese bei der Modularisierung zu Aspekten führen, muss der Begriff der "Kompatibilität einer Schnittstelle" überdacht werden. In herkömmlichen Systemen wird die Schnittstelle eines Moduls im Wesentlichen durch Funktionssignaturen und verwendete Datentypen beschrieben. Durch Aspekte existieren nun noch die in Abschnitt 5.1.4 und 5.1.5 beschriebenen Einwirkungs- und Ordnungsbeziehungen.

Bei den Beziehungen von Aspekten, die auf keiner Seite "notwendig" sind, ist die Kompatibilitätsfrage relativ leicht zu beantworten. Wenn sich zum Beispiel ein *Crosscutting Concern* auf eine Gruppe von Funktionen auswirkt, so können diese Funktionen aus Sicht dieser Beziehung ohne weiteres ausgetauscht werden. Gleiches gilt für die nicht notwendige Ordnungsbeziehung.

Bezüglich der "notwendigen" Beziehungstypen sieht die Situation problematischer aus. Tabelle 5.1 auf der nächsten Seite stellt dar, was Kompatibilität in Bezug auf den jeweiligen Beziehungstyp bedeutet. Wie zu erkennen ist, geht es immer wieder um die Frage, ob bestimmte "Auswirkungen" hinreichend sind, damit ein anderes Modul korrekt arbeiten kann. Ob eine solche Bedingung erfüllt ist, ist keineswegs leicht zu sehen oder automatisch zu bestimmen. Daher lässt diese Übersicht den Schluss zu, dass die notwendigen Aspektbeziehungen besser zu vermeiden sind. Eine falsche Beurteilung dieser nicht trivialen Kompatiblitätseigenschaft durch beispielsweise den Entwickler kann leicht zu nicht funktionierenden Systemkonfigurationen führen.

Abschließend sei noch bemerkt, das der Name "funktionale Hierarchie" für eine solche Struktur, die ganz allgemein *Concerns*, das heißt *Crosscutting Concerns* und herkömmliche Funktionen, beschreibt, nicht angebracht ist. Daher wird dafür fortan der Begriff "*Concern*-Hierarchie" verwendet.

Beziehungsart	Kompatibilität liegt vor, wenn ...
Einwirken (notwendig für Komponenten)	die durch die Einwirkung (zum Beispiel Ausführung von Aspektcode) ausgelösten Zustandsänderungen hinreichend sind, damit die Komponente korrekt arbeiten kann.
Einwirken (notwendig für Aspekt)	die Komponente, auf die der Aspekt wirken soll, ein entsprechendes Verhalten und eine Menge von Verbindungspunkten aufweist, so dass der einwirkende Aspekt korrekt arbeiten kann.
notwendige Ordnung	die Auswirkungen des Aspekts hinreichend sind, damit der Aspekt, der auf die Ordnung angewiesen ist, korrekt arbeiten kann.

Tabelle 5.1: Kompatibilität bei Beziehungen von Aspekten

5.2 Unterstützung des Implementierungsprozesses

In den letzten Abschnitten wurden die nötigsten konzeptionellen Grundlagen beschrieben, um Aspektorientierung und Programmfamilien beim Systementwurf in Einklang zu bringen. Eine Untersuchung der Vor- und Nachteile des hier verfolgten Ansatzes, Aspektorientierung und Programmfamilien für den Bau von effizienten Spezialzweckbetriebssystemen zu kombinieren, erfordert jedoch eine Umsetzung. Diese Umsetzung besteht in der aspektorientierten Implementierung mehrerer *Crosscutting Concerns* in einer Betriebssystemfamilie. Vorher ist es jedoch erforderlich, zu benutzende Werkzeuge und Bewertungskriterien festzulegen.

5.2.1 Arten von Crosscutting Concerns

Aspektorientiert zu programmieren, bedeutet die Implementierung von *Crosscutting Concerns* von den eigentlichen Systemkomponenten zu trennen und die sonst verstreuten Implementierungsfragmente an einer Stelle zu bündeln. Dazu soll nach Möglichkeit eine problemadäquate Sprache verwendet werden. Das bedeutet nicht zwangsläufig, dass wie bei AspectJ Komponentencode und Aspektcode in derselben Sprache verfasst werden. Dennoch hat diese gemeinsame Sprachbasis viele Vorteile. Beispielsweise können Aspekte und Klassen in AspectJ dieselben Datentypen und Objekte benutzen. Desweiteren ist das Lernen der Sprache, nachdem die Einstiegshürde, die Idee von AOP zu verstehen, überwunden wurde, verhältnismäßig leicht. Daher ist es wünschenswert, möglichst viele Aspekte mit Hilfe einer solchen aspektorientierten Spracherweiterung zu implementieren.

Daneben gibt es jedoch immer noch andere Arten von *Crosscutting Concerns*, die mit einer solchen Vielzweckspracherweiterung nicht modular implementiert werden können. So beschränken sich die Eingriffsmöglichkeiten von AspectJ bei-

spielsweise auf Verbindungspunkte, bei denen Attribute gelesen oder geschrieben werden sowie solche, wo Funktionen aufgerufen oder ausgeführt werden und noch wenige andere. Daneben können mit den sogenannten *Introductions* Klassen um Attribute erweitert werden. Aspekte, die auf andere Arten von Verbindungspunkten wirken, andere Arten von Aktivierungen oder weitergehende Informationen über potentielle Verbindungspunkte erfordern, sind nicht implementierbar. Beispielsweise kann mit AspectJ kein Protokollierungsaspekt implementiert werden, der ausgeführten Code auf der Ausdrucksebene aufzeichnet.

Eine zweite Art von *Crosscutting Concerns*, die durch eine Spracherweiterung wie AspectJ nicht unterstützt wird, sind solche, die komplexe Programmrestrukturierungen nötig machen. So wurde in einem der ersten Papiere über AOP [67] das Beispiel eines Lisp Programms genannt, das zur Bildmanipulation eingesetzt wird. Dieses wurde aus Optimierungsgründen durch ein zweites in Lisp geschriebenes Aspektprogramm komplett umstrukturiert. Die Eingabe dieses Metaprogramms war der Aufrufgraph des Komponentencodes, der wie eine gewöhnliche Datenstruktur manipuliert werden konnte. Aufgrund dieser Datenstrukturmanipulation wurde vom Aspektweber eine Programmmanipulation durchgeführt. Das Ergebnis war, dass der Komponentencode eine saubere und verständliche Struktur hatte und dass trotzdem aufgrund des Optimierungsaspekts eine gute Rechenleistung erzielt werden konnte. Solche Beispiele sind natürlich auch auf andere Sprachen übertragbar.

Zusammenfassend kann man zwischen drei Arten von *Crosscutting Concerns* unterscheiden:

1. *Concerns*, die auf der Ebene der Komponentensprache durch eine aspektorientierte Spracherweiterung abgedeckt werden können, die Reaktionen auf Programmzustände und das Ergänzen des Programms um Typen und Attribute erlaubt.

2. Solche, die so spezielle Verbindungspunkte betreffen bzw. so spezielle Informationen über Verbindungspunkte benötigen, dass eine allgemeine aspektorientierte Spracherweiterung diese Fälle nicht abdecken kann.

3. *Crosscutting Concerns*, die maßgeblich die Struktur eines Programm bestimmen, so dass eine aspektorientierte Implementierung komplexe Manipulationen erfordert.

Ein großer Teil der Forscher, Entwickler und Anwender aus dem AOP Bereich, versteht unter aspektorientierter Programmierung lediglich die modulare Implementierung der ersten Gruppe von *Crosscutting Concerns*. Der Grund besteht darin, dass AspectJ mit seinem Bekanntheitsgrad die allgemeine Meinung über die Bedeutung der Begriffe prägt. Aus diesem Grund soll im Zusammenhang mit der zweiten und dritten Gruppe von *Concerns* fortan von "aspektorientierten Ansätzen zur Implementierung" und nicht von "aspektorientierter Programmierung" gesprochen werden. Dieser Begriff bleibt aspektorientierten Spracherweiterungen im Sinne von AspectJ vorhalten. Die Idee ist dennoch dieselbe.

5.2.2 Notwendige Werkzeuge oder Sprachen

Über den wirklichen Wert des Einsatzes von AOP beim Entwurf, der Implementierung und der Wartung einer Betriebssystemfamilie kann man an dieser Stelle nur spekulieren. Die Menge der Konfigurationspunkte zu reduzieren und globale Strategien konfigurierbar gestalten zu können, ist gerade in diesem Bereich eine reizvolle Aussicht. Ob diese Effekte wirklich eintreten, kann nur eine Umsetzung der vorgestellten Konzepte zeigen.

Aspektorientiert zu programmieren bedeutet, dass mindestens ein Werkzeug, nämlich der Aspektweber, vorhanden sein muss. Dieser Weber kann wie bei AspectJ zur Übersetzungszeit arbeiten und Codetransformationen vornehmen oder wie im AOSA Rahmenwerk (siehe Abschnitt 3.3.2) zur Laufzeit arbeiten. Weitere Werkzeuge wie Entwicklungsumgebungen, die anzeigen können, welche Aspekte an welchen Stellen im System wirken, wären hilfreich, haben jedoch nicht höchste Priorität[4].

In der Zusammenfassung von Kapitel 3 wurde bereits gesagt, dass zur Zeit keine aspektorientierte Programmiersprache oder -spracherweiterung existiert, die für das Einsatzgebiet Betriebssystemfamilien geeignet ist. Insbesondere gilt das, wenn beachtet wird, dass solche Familien besonders im Bereich kleinster eingebetteter Systeme sinnvoll sind. Die verwendeten Werkzeuge oder die verwendete Programmiersprache sollten diesen Einsatzbereich auf keinen Fall ausschließen.

Die Betriebssystemfamilie PURE wurde mit Hilfe der Programmiersprache C++ erstellt. Dies hat gute Gründe. So erreicht man mit C++ ein hohes Maß an Portabilität, hat abstrakte Datentypen und Vererbung, die gut für die Umsetzung eines familienbasierten Entwurfs genutzt werden kann. Gleichzeitig hat sich gezeigt, dass bei vorsichtiger Verwendung der "teuren" C++ Spracheigenschaften, wie dem dynamischen Binden oder Templates, ein mit C vergleichbar kompakter Code erzeugt wird. Aus diesem Grund wäre eine aspektorientierte Spracherweiterung für C++ ein sinnvoller Ausgangspunkt für Untersuchungen über *Crosscutting Concerns* im Betriebssystemumfeld. Wie im letzten Abschnitt beschrieben wurde, kann jedoch eine solche Spracherweiterung nicht alle denkbaren Arten von *Crosscutting Concerns* abdecken. Sie sollte daher auf einem Codeanalyse- und -manipulationssystem für C++ aufsetzen, das auch von anderen (Spezial-) Aspektwebern genutzt werden kann.

In den folgenden Kapiteln wird nun zuerst der Entwurf und die Implementierung eines als Basis dienenden Codeanalyse- und Codetransformationssystems namens PUMA[5] beschrieben. Darauf folgt die Beschreibung von AspectC++, einer aspektorientierten C++ Spracherweiterung, und weiterer auf PUMA basierender Aspektweber.

[4] Für AspectJ existieren zu diesem Zweck bereits mehrere Erweiterungen für Entwicklungsumgebungen, siehe `http://aspectj.org/`.

[5] PURE Manipulator

5.3 Bewertung

Um den Wert des Ansatzes und der implementierten Werkzeuge abschätzen zu können, werden in Kapitel 9 mehrere Fallstudien im Kontext von PURE betrachtet. Dadurch sollen folgende Fragen beantwortet werden:

1. Sind die vorgeschlagenen Modellerweiterungen geeignet?
2. Funktionieren alle entwickelten Werkzeuge?
3. Ist es durch AOP nun möglich, globale Strategien im Betriebssystem modular und damit leicht konfigurierbar zu implementieren?
4. Reduziert sich durch aspektorientierte Ansätze die Anzahl der sichtbaren Konfigurationspunkte in einer Betriebssystemfamilie?
5. Benötigt man für eine aspektorientierte Implementierung eines *Crosscutting Concerns* weniger Quellcode?
6. Lassen sich Aspekte im Betriebssystemcode ohne zusätzlichen Ressourcenverbrauch realisieren?

Auch durchweg positive Antworten auf diese Fragen würden nicht den Schluss zulassen, dass die neuen Techniken automatisch zu besser wartbarem Code oder kürzeren Entwicklungszeiten führen. Ersteres ist sehr schwer zu beurteilen. Hier kann nur auf Basis von Indizien argumentiert werden, solange man nicht den Wartungsaufwand für mehrere Paare von funktional und qualitätiv gleichwertigen Softwaresystemen über Jahre beobachtet, von denen eins aspektorientiert und das andere mit herkömmlichen Entwicklungstechniken erstellt wurde. Indizien, die für einen geringeren Wartungsaufwand sprechen würden, wären jedoch durch die Punkte 3 bis 5 gegeben.

Kürzere Entwicklungszeiten könnte man prinzipiell durch Parallelentwicklungsstudien zeigen. Dies wäre jedoch unter Verwendung von neuen Werkzeugen und für viele Entwickler neuen Konzepten in Konkurrenz zu jahrzehnte alten Programmierparadigmen und bewährten Sprachen ein unfaires Duell. Solche Studien können sinnvoll erst in einigen Jahren durchgeführt werden[6].

5.4 Zusammenfassung

Aspekte haben typischerweise eine lose Bindung zu dem Code, auf den sie einwirken. Das ist ein großer Vorteil für die Konfigurierbarkeit einer Programmfamilie, da beide Seiten so problemlos herauskonfiguriert werden können, ohne dass die

[6] Dies zeigt auch, dass die von Gail Murphy durchgeführten Studien [113] laut einem kürzlich erschienenen Kommentar von Gregor Kiczales auf der AspectJ Mailing Liste als veraltet eingestuft werden, da sich die Sprache und die Verlässlichkeit der Implementierung inzwischen deutlich weiter entwickelt haben.

andere in ihrer Arbeit beinträchtigt wird. Anders sieht die Situation beim notwendigen Einwirken und der notwendigen Ordnung aus. Wie erläutert wurde, haben diese Beziehungstypen ein Risikopotential, so dass sie nicht oder nur wenig eingesetzt werden sollten. Die Fallstudien in Kapitel 9 werden zeigen, ob man darauf verzichten kann.

Weiterhin wurde gezeigt, dass sich das Modell der funktionalen Hierarchie um *Crosscutting Concerns* erweitern lässt. Aufgrund der modularen Implementierbarkeit von *Crosscutting Concerns* durch Aspekte können sie jetzt weitgehend wie gewöhnliche Funktionen behandelt werden.

Den Abschluss des Kapitels bildete eine Erörterung des weiteren Vorgehens. Dabei wurden Bewertungskriterien aufgestellt und notwendige Werkzeuge festgelegt. Wichtig dabei ist, dass *Crosscutting Concerns* nicht nur auf der Ebene der Spracherweiterung AspectC++ behandelt werden sollen, sondern auch andere Arten abgedeckt werden. Dadurch soll zum Beispiel erreicht werden, dass auch die in 4.2.3 erörterte Problematik der Implementierungsstrukturen mittels eines aspektorientierten Ansatzes gelöst wird.

Nicht angesprochen wurde in diesem Kapitel die Frage der Verifizierbarkeit aspektorientierter Programme. Softwareverifikation ist generell eine schwierige Aufgabe und die Konstrukte einer aspektorientierten Programmiersprache haben, was den Kontrollfluss angeht, im Vergleich zu herkömmlichen funktionalen, logischen, imperativen oder objektorientierten Sprachen eine recht komplizierte Semantik. Da die Frage der Verifizierbarkeit und auch die des Testens solcher Programme nicht in einem direkten Zusammenhang zu Betriebssystemen und Programmfamilien steht, wird in dieser Arbeit auf eine tiefgehende Betrachtung der damit zusammenhängenden Probleme verzichtet. Einige erste Ideen zu diesem Themenkomplex werden aber trotzdem in Kapitel 10 vorgestellt.

Kapitel 6

PUMA

PUMA ist die Abkürzung für PURE Manipulator. Der primäre Zweck dieses Werkzeugs ist die Manipulation von PURE Betriebssystemcode im Rahmen der Fallstudien, die in Kapitel 9 beschrieben werden. Die hier dokumentierte Entwicklung ist jedoch von vornherein keineswegs nur darauf beschränkt gewesen. Es handelt sich um ein allgemein verwendbares Analyse- und Manipulationssystem für C++ Programme.

6.1 Anforderungen

Das Weben von Aspekten erfordert unter Umständen genaue Kenntnisse der Programmvariablen und des Kontrollflusses. Zudem sollen komplexe Codetransformationen ausgeführt werden können. Damit kommt ein Weben von Binärcode nicht in Frage. PUMA muss stattdessen die Fähigkeit der Analyse von C++ Quelltexten aufweisen. Auf der Ausgabeseite sollte es möglich sein, diverse Zielplattformen zu unterstützen. Um dieses Ziel zu erreichen, ist im Rahmen dieser Arbeit die Auslegung von PUMA als Quellcodetransformationssystem der einzig praktikable Weg. Auf diese Weise können die Dienste existierender Übersetzer genutzt werden und PUMA selbst kann zielplattformunabhängig ausgelegt werden.

Eine aspektorientierte Spracherweiterung wie das im nächsten Kapitel vorgestellte AspectC++ benötigt weitreichende semantische Informationen über den Code, auf den ein Aspekt wirkt. In AspectJ kann beispielsweise *Advice*-Code für Aufrufe von bestimmten Funktionen definiert werden. Die Menge dieser Funktionen wird über Namensmuster in einer *Pointcut*-Definition festgelegt. Auch für eine aspektorientierte C++ Erweiterung wäre dieses Feature sinnvoll. Das bedeutet, dass das zugrunde liegende Codeanalysesystem weitreichende semantische Informationen bereitstellen muss. Unter anderem müssen alle Funktionsaufrufe aufgelöst werden, was bei C++ aufgrund der Möglichkeit des Überladens von Funktionen und der diversen Sichtbarkeitsregeln keine leichte Aufgabe ist [58]. Dieser Punkt und die im nächsten Abschnitt näher erläuterte Tatsache, dass allein für das syntaktische Verständnis von C++ eine weitreichende semantische Analyse erforderlich

ist, sind dafür verantwortlich, dass PUMA eine solche semantische Analysefunktion bereitstellen muss. Damit muss PUMA wie ein *Compiler Front End* mindestens aus den drei Komponenten lexikalische-, syntaktische- und semantische Analyse bestehen.

Eine andere wichtige Anforderung ergibt sich aus der Struktur von C++ Quelltexten. Sie bestehen zum einen aus den Kernsprachelementen von C++ und zum anderen aus Präprozessordirektiven. Diese Direktiven werden hauptsächlich zum Definieren von Makros und zum Einbinden von Dateiinhalten benutzt. Mit diesem aus C übernommenen Konzept können die Schnittstellen anderer Übersetzungseinheiten bekannt gemacht werden, so dass Aufrufe externer Funktionen vom Übersetzer überprüft und erzeugt werden können. Dieser Mechanismus wird auch für die Schnittstelle von Bibliotheken benutzt. Bibliotheken bestehen bei C++ aus sogenannten *Header*-Dateien, die die Schnittstelle in Form von Quellcode beschreiben, und einer Bibliotheksdatei, die den Binärcode der Funktionen enthält. Für ein Codemanipulationssystem stellt sich dadurch die Frage, ob Präprozessordirektiven als Eingabe gestattet und damit als Teile der Eingabesprache angesehen werden, oder ob die Umgebung dafür sorgen muss, dass der Präprozessorlauf bereits durchgeführt wurde. Letzteres wäre leichter zu implementieren, verhindert jedoch einige mögliche Anwendungen:

- die Analyse und Manipulation von Präprozessordirektiven durch Aspektweber
- das Annotieren von C++ Konstrukten durch spezielle Kommentare, die durch Aspektweber ausgewertet werden[1]
- das Bereitstellen manipulierten Quellcodes als Bibliothek

Gerade der letzte Punkt ist für die PURE Entwickung von Bedeutung. Wenn er nicht erfüllt wäre, wären Anwender des als Bibliothek vorliegenden Betriebssystems dazu gezwungen, nicht ihre bevorzugte Programmiersprache, sondern grundsätzlich AspectC++ und eventuell noch weitere Aspektweber zu benutzen. Daher soll PUMA einen integrierten C Präprozessor enthalten und auf Wunsch auch manipulierte *Header*-Dateien ausgeben können.

Die nächste große Anforderung besteht darin, dass PUMA eine anwenderfreundliche Programmierschnittstelle zur Verfügung stellen muss, die alle erforderlichen Analyse- und Manipulationsfunktionen zugänglich macht, aber keine weiteren Kenntnisse über die interne Struktur von PUMA oder etwa Details der C++ Syntax verlangt. Anwenderfreundlichkeit schließt insbesondere auch die leichte Erweiterbarkeit ein. So wird AspectC++ als aspektorientierte Variante von C++ eine erweiterte Syntax und Semantik aufweisen, deren Implementierung nach Möglichkeit auf der von C++ (innerhalb von PUMA) basieren soll.

[1] Kommentare werden gewöhnlich wie Direktiven durch den Präprozessor entfernt.

Obwohl es bei den zu manipulierenden Systemen wie Pure auf höchste Effizienz ankommt, ist dieser Gesichtspunkt bei der Entwicklung von Puma weniger bedeutend. Eine statische Konfigurierbarkeit, die zu administrativem Aufwand für die Anwender führt, wäre eher belastend als hilfreich, da davon ausgegangen werden kann, dass für die Übersetzung eines Softwaresystems grundsätzlich ein leistungsfähiger Rechner verfügbar ist.

6.2 Probleme

Aus den im letzten Abschnitt beschriebenen Anforderungen an Puma ergibt sich eine Reihe von Problemen, die näher betrachtet werden sollen.

6.2.1 Analyse von C++ Code

Die Analyse von C++ Code ist eine ausgesprochen komplexe Aufgabe. An dieser Stelle sollen nur die wesentlichen Probleme aufgezeigt werden, die diese Aufgabe mit sich bringt. Detailliertere Erläuterungen sind einer Diplomarbeit zu entnehmen, die die Parser Entwicklung von Puma zum Inhalt hatte [110].

Hauptursache aller Schwierigkeiten bei der Analyse von C++ Code ist, dass die Grammatik eine kontextabhängige Unterscheidung von Namen (engl. *identifier*) erfordert. Namen sind lexikalische Einheiten, die im einfachsten Fall im Rahmen der lexikalischen Analyse erkannt werden, und von dort direkt an die Syntaxanalyse weitergereicht werden. Bei C++ dagegen muss vor der Syntaxanalyse jeder Name untersucht werden, ob er eventuell ein schon bekannter Name einer Typdefinition (*typedef-name*[2]), eines Namensraums (*namespace-name, original-namespace-name, namespace-alias*), einer Klasse (*class-name*), eines Aufzählungstyps (*enum-name*) oder eines Templates (*template-name*) ist. Damit müssen schon in einer sehr frühen Phase der Programmanalyse Operationen durchgeführt werden, die im Compilerbau nach der "reinen Lehre" erst nach der Syntaxanalyse im Rahmen der semantischen Analyse stattfinden sollten [1]. Diese Schwierigkeit bestand bereits bei C und wurde dort in der Regel durch den sogenannten "lexical typedef hack", das heißt dem Nachschlagen jedes Namens in Symboltabellen vor der Weitergabe an die Syntaxanalyse, gelöst. In C++ ist dies jedoch nicht so einfach. Zum einen gibt es qualifizierte Namen wie `A::B::x` oder auch `A<10+sizeof(int)>::B<float>::x`, so dass man gezwungen wird, schon während der Syntaxanalyse den Wert von Ausdrücken, ihren Typ und die Auswahl von *Template*-Spezialisierungen zu bestimmen. Sogar die Größe von Objekten, die stark von der Codegenerierung und der Zielplattform abhängt, muss bekannt sein. Darüber hinaus ergeben sich bei dieser Vorgehensweise schnell Synchronisationsprobleme zwischen dem Parser und der vorgeschalteten Komponente zum Bewerten

[2] Die Bezeichnungen *typedef-name*, *namespace-name*, u.s.w. beziehen sich auf die C++ Grammatik, wie sie von der ISO festgelegt wurde [58].

von Namen, wenn der Parser vorausschauend (*lookahead*) oder nach dem *Backtracking*-Prinzip[3] arbeitet. Dieses und weitere Probleme mit der Mehrdeutigkeit der C++ Syntax sind dafür verantwortlich, dass ein hohes Maß an Kontrolle über den Syntaxanalysevorgang erforderlich ist. Zudem muss bereits ein sehr großer Teil der semantischen Analyse gleichzeitig ablaufen.

6.2.2 Erweiterbarkeit der Implementierung

PUMA wurde als Analyse- und Manipulationssystem für C++ Code konzipiert. Das wirft die Frage auf, ob PUMA überhaupt als Basis für die Implementierung der Sprache AspectC++ dienen kann. Schließlich ist zu erwarten, dass diese Sprache eine veränderte Syntax und Semantik besitzen wird. Das System sollte daher erweiterbar gestaltet sein, so dass wenigstens ein großer Teil der C++ Implementierung für AspectC++ wiederverwendet werden kann.

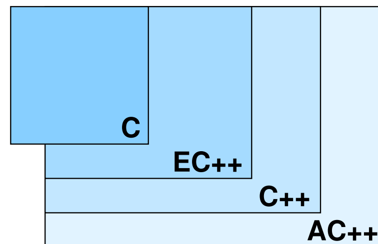


Abbildung 6.1: Die Familie der C-basierten Sprachen

C++ und AspectC++ sind nicht die einzigen von C abgeleiteten Sprachen. Dies veranschaulicht Abbildung 6.1. Sie zeigt neben C, C++ und AspectC++ noch EC++ [106], was für *Embedded C++* steht[4]. Daneben ist zu sehen, dass C einige Spracheigenschaften besitzt, die in den C++ Sprachen nicht zur Verfügung stehen. Es handelt sich dabei um Erweiterungen durch den neuesten ISO Standard [59], einige spezielle Konstrukte des ursprünglichen K&R C [64] und vor allem um Unterschiede in der Sprachsemantik.

Durch eine erweiterbare Implementierung der syntaktischen und semantischen Analyse wäre man in der Lage, mit PUMA all diese Sprachen ohne Coderedundanz abzudecken, wobei AspectC++ eine externe und die drei anderen interne Erweiterungen wären. Dies ist jedoch schwer zu bewerkstelligen, da die klassischen Compilergenerierungswerkzeuge wie Yacc [61] die Erweiterung eines fertigen binären

[3] Bei diesem Prinzip läuft der Parser zunächst “auf gut Glück” in eine Syntax-Regel hinein, um dann im Fehlerfall den vorherigen Zustand wieder herzustellen und – sofern vorhanden – die nächste Regel zu probieren.

[4] Bezüglich der Spracheigenschaften ist EC++ zwischen C und C++ angesiedelt. Dieser Standard ist erst nach C++ entstanden, um speziell Programmierern im Bereich der eingebetteten Systeme eine einfache objektorientierte Sprache zu geben, die teure oder fehlerträchtige Sprachelemente von C++ vermeidet. So wurde unter anderem auf *Templates*, *Run-Time Type Information* (RTTI), rein-virtuelle Funktionen und Mehrfachvererbung verzichtet.

Parsers nicht vorsehen. Zwar unterstützt mindestens der Parser Generator ANTLR[5] das Erben von Grammatiken (*grammar inheritance* [2]), doch dies führt letztlich nur zu einem Kopieren der Regeln und damit zum Generieren mehrerer Parser mit starker Redundanz. Ohne einen passenden Codegenerator muss der Parser für die Grammatiken von Hand implementiert werden.

6.2.3 Integration des Präprozessors

Die Integration eines C Präprozessors und insbesondere die Möglichkeit, nach einer Manipulation wieder *Header*-Dateien auszugeben, bringt einige Schwierigkeiten mit sich. So müssen die Informationen über die ursprüngliche Dateistruktur beim Einlesen einer Übersetzungseinheit erhalten bleiben. Damit muss neben dem eigentlichen Syntaxbaum noch ein Präprozessor Syntaxbaum aufgebaut und erhalten bleiben, und beide sind geeignet zu verbinden.

Durch das Einbinden von *Header*-Dateien in verschiedenen Übersetzungseinheiten kommt es zu dem Effekt, dass die gleichen Programmfragmente sich in mehreren Syntaxbäumen gleichzeitig repräsentiert finden. Dies kann zu Problemen bei der Programmmanipulation führen. Soll zum Beispiel eine bestimmte Variable in allen Übersetzungseinheiten umbenannt werden, kann es dazu kommen, dass mehrfach versucht wird, dasselbe Codefragment zu manipulieren. Dies muss unbedingt vermieden werden.

Problematisch ist auch die Behandlung von Makros. Das folgende Codefragment zeigt die Definition eines solchen Präprozessormakros und eine darauf folgende zweifache Anwendung:

```
#define TRIGOFUNC(name) float name (float)
TRIGOFUNC(sinus);
TRIGOFUNC(cosinus);
```

Die mit “#define” beginnende Makrodefinition muss vom Präprozessor erkannt und gespeichert werden. Alle im Programmtext folgenden Namen sind vor der Weitergabe an den eigentlichen C++ Parser mit den Namen bekannter Makros zu vergleichen. Ist der Name gleich, so wird eine Makroexpansion unter Berücksichtigung eventueller Makroparameter durchgeführt. Aus

```
TRIGOFUNC(sinus);
```

würde dann

```
float sinus (float);
```

werden. Da der C++ Parser nur noch das expandierte Makro zu sehen bekommt, wird auch der Syntaxbaum genau dieses repräsentieren. Wenn nun durch die Anwendung die Anweisung kommt, beispielsweise den Argument- und Rückgabetyp der Funktion `sinus` von `float` auf `double` zu ändern, steht PUMA vor einer

[5] Another Tool for Language Recognition, Teil des Purdue Compiler Construction Tool Set (PCCTS) [90].

schwierigen Entscheidung. Ursprünglich kommen die beiden `float` Typbezeichner aus der Makrodefinition. Diese dort zu manipulieren würde jedoch auch Auswirkungen auf andere Makroexpansionen haben. Wäre das gewünscht, sollte die Anwendung eher die Makrodefinition direkt manipulieren. Damit bleibt nur noch die Wahl zwischen dem Zurückweisen der Manipulation und dem Expandieren des Makroaufrufs im Quelltext mit anschließender Manipulation. Damit wäre die Makrodefinition von der Änderung nicht betroffen und der Wunsch der Anwendung könnte erfüllt werden. Aber auch diese Lösung ist nicht perfekt, da nachfolgende Änderungen der Makrodefinition dann keine Auswirkungen auf die Deklaration der Sinus-Funktion mehr hätten. Deshalb wurde festgelegt, dass Manipulationen an Makro-generiertem Code zurückgewiesen werden.

6.3 Entwurf

Dieser Abschnitt beschreibt die Entwurfsentscheidungen für die PUMA Entwicklung, die aus den in den vorangegangenen Abschnitten beschriebenen Anforderungen und Problemen resultieren.

6.3.1 Schichtenstruktur

Die grobe Schichtenstruktur des PUMA Systems ist in Abbildung 6.2 auf der nächsten Seite dargestellt. Jede Ebene basiert jeweils auf den darunter liegenden Ebenen. Ihre Bedeutung ist wie folgt:

Quelltextebene Diese Ebene ist für die Verwaltung der sogenannten *Units* zuständig. Das sind Objekte, die Quelltextdateien repräsentieren. *Header*-Dateien und vollständige Übersetzungseinheiten werden auf dieser Ebene gleich behandelt. Dazu kommen noch Funktionen, die das Lesen und Schreiben von Dateien ermöglichen, so dass diese Ebene alle von Portierungen betroffenen Teile des PUMA Systems kapselt[6].

Scannerebene Der Begriff *Scanner* wird für die Codeanalysekomponente verwendet, die die lexikalische Analyse, das heißt das Zerlegen eines Zeichenstroms in sogenannte *Tokens*, durchführt. Hier werden zum Beispiel Schlüsselworte, Kommentare oder Präprozessordirektiven als solche erkannt und gespeichert. Das Ergebnis sind Listen von *Tokens* pro *Unit*, die den Inhalt der jeweiligen Datei repräsentieren. Diese Repräsentation erfolgt verlustfrei. Das heißt, dass beispielsweise auch Folgen von Leerzeichen oder Zeilenumbrüchen als *Tokens* gespeichert werden. Dadurch kann eine manipulierte *Unit* später unter Erhaltung aller Formatierungen wieder ausgegeben werden und bleibt notfalls auch für Menschen lesbar. Abbildung 6.3 auf der nächsten Seite verdeutlicht die Arbeitsweise der Scannerebene. Sie zeigt auf der

[6]PUMA läuft derzeit unter Linux, Solaris, MacOS X und Windows. Es werden CPUs der x86, Alpha, Sparc und PowerPC Familie abgedeckt.

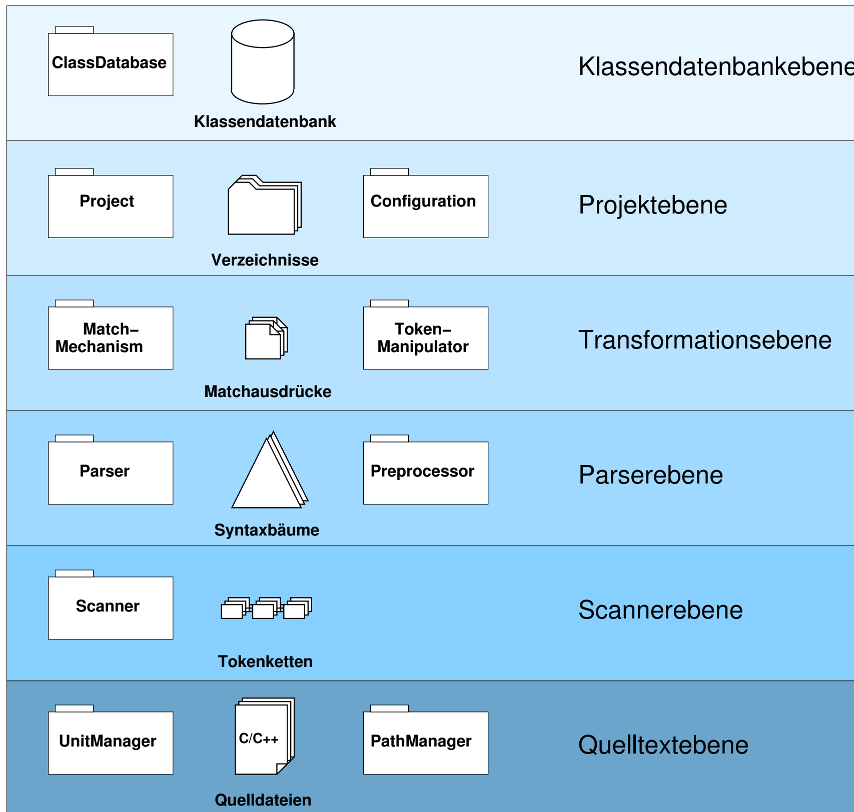


Abbildung 6.2: Die Schichtenstruktur des PUMA Systems

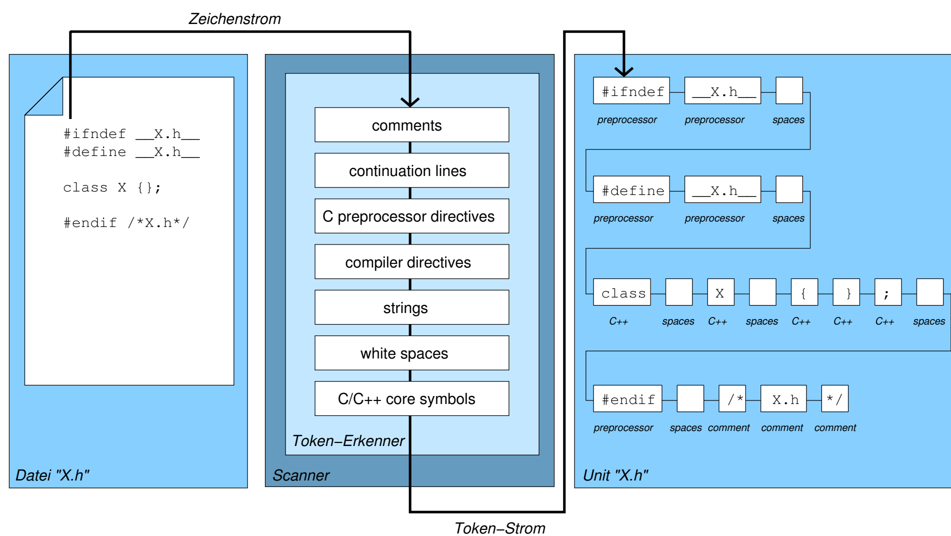


Abbildung 6.3: Arbeitsweise der Scannerebene

linken Seite den eingelesenen Quelltext und auf der rechten Seite die einer *Unit* zugeordnete resultierende Liste von *Tokens*. Die *Scanner*-Komponente selbst besteht aus verschiedenen priorisierten *Token*-Erkennern für die verschiedenen Arten von *Tokens*. Der Code der *Token*-Erkenner wird generiert und implementiert jeweils einen minimalen deterministischen endlichen Automaten.

Parserebene Hier wird anhand der *Tokens*, die zu einer Übersetzungseinheit gehören, eine syntaktische und semantische Analyse durchgeführt. Um diese *Tokens* zu ermitteln, muss vor dem eigentlichen C++ *Parser* ein Präprozessor-*Parser* laufen und Direktiven wie "#include" zum Einbinden von *Header*-Dateien sowie Makros und Direktiven zur bedingten Übersetzung auswerten. Wichtig ist dabei, dass die erstellten Syntaxbäume Zeiger auf die entsprechenden *Tokens* enthalten, so dass höhere Ebenen später zwecks Manipulation von den syntaktischen Strukturen auf *Tokens*, Dateien und Positionen im Quelltext schließen können. Abbildung 6.4 zeigt diese Verbindung zwischen

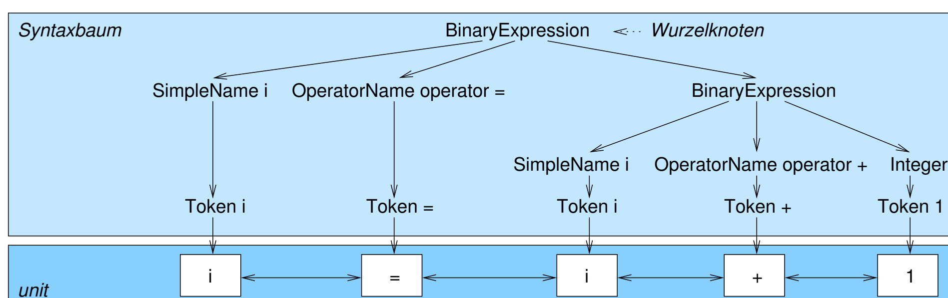


Abbildung 6.4: Ein Syntaxbaum und seine Verbindungen zu *Tokens*

einem Syntaxbaum und einer *Unit*. Weitere Details zu dieser Ebene werden in Abschnitt 6.3.2 präsentiert.

Transformationsebene Diese Ebene stellt Operationen zur Analyse von Syntaxbäumen und zur Codemanipulation bereit. Wichtig ist dabei, dass der Anwendungsprogrammierer keine Details über die Repräsentation von Syntaxbäumen kennen muss. Dadurch soll erreicht werden, dass PUMA leicht zu benutzen ist und dass Anwendungsprogramme von Änderungen an diesen Strukturen unabhängig sind. Um dies zu gewährleisten, wurde der sogenannte *Match*-Mechanismus entworfen und implementiert, der zusammen mit dem hier verfolgten Transaktionskonzept in den Abschnitten 6.3.3 und 6.3.4 näher erläutert wird.

Projektebene Während die bisher beschriebenen Ebenen lediglich eine Sicht auf einzelne Dateien und Übersetzungseinheiten hatten, dient die Projektebene der Beschreibung aller zu einem Projekt gehörenden Dateien. Dazu zählen

neben den Pfaden für Ein- und Ausgabeverzeichnisse bei Transformationen auch Konfigurierungsinformationen für die *Parser*. So kann hier projektweit festgelegt werden, in welcher Programmiersprache die Quelltexte verfasst sind und welche übersetzerspezifischen Erweiterungen eventuell genutzt wurden. Desweiteren ist es hier möglich, *Include*-Pfade für den Präprozessor, vordefinierte Makros und vieles mehr festzulegen.

Klassendatenbankebene Die sogenannte Klassendatenbank von PUMA ist eine Datenstruktur, die praktisch alle aus der semantischen Analyse resultierenden Informationen den Anwendungsprogrammen zugänglich macht. Beispielsweise kann hier die Liste aller Klassen inklusive Methoden und Attributen und deren Typen erfragt werden. Ein Beispiel, das zeigt, wie eine Anwendung über die Klassendatenbank Informationen über einen C++ Quelltext erhält, wird in Abschnitt 6.5 vorgestellt. Wichtig an dieser Ebene ist, dass beim Aufbau der Datenstruktur die Funktion des Binders nachgeahmt wird. So wird eine in mehreren Übersetzungseinheiten deklarierte Klasse, Funktion oder Variable gegebenenfalls als ein Objekt erkannt, so dass in der Klassendatenbank dafür auch nur ein Eintrag angelegt wird. Dadurch kann das Problem vermieden werden, dass Anwendungen versuchen, die in einer *Header*-Datei befindliche Deklaration mehrfach zu manipulieren.

6.3.2 Die *Parser*-Familie

Um ein hohes Maß an Wiederverwendbarkeit und auch Konfigurierbarkeit zu erreichen, wurde die Codeanalyse für die Familie der C-basierten Sprachen (siehe Abbildung 6.1 auf Seite 70) ebenfalls als Familie konzipiert. Abbildung 6.5 zeigt

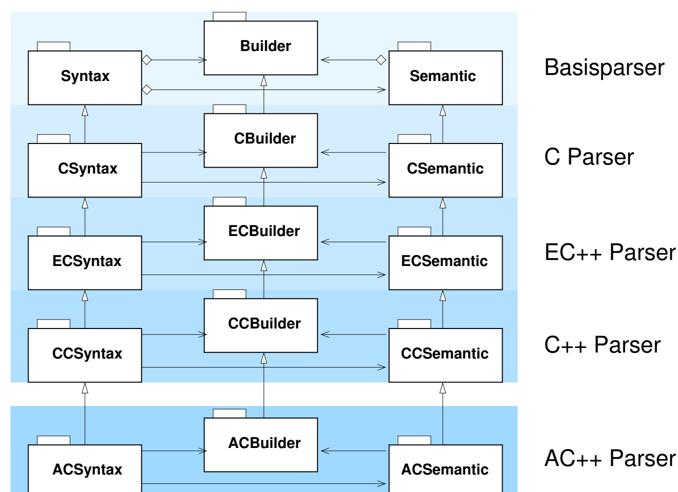


Abbildung 6.5: Die Struktur der *Parser*-Familie für PUMA

die entsprechende Klassenstruktur, mit der *Parser* für vier Sprachen instanziiert werden können. Davon sollen C, EC++ und C++ Teil von PUMA sein und AC++ wird im Rahmen der AspectC++ Implementierung unter Wiederverwendung des C++ *Parsers* erstellt. Wie zu erkennen ist, wird hier wie bei PURE der Vererbungsmechanismus zur Erweiterung eingesetzt.

Die Aufteilung in `Syntax`, `Builder` und `Semantic` Klassen dient der weiteren Vereinfachung der Module, was bei dieser hochkomplexen Aufgabe unbedingt notwendig ist. So wird erreicht, dass die `Syntax` Klassen im wesentlichen nur die aus der Grammatik entnommenen Regeln implementieren müssen. Die `Builder` Klassen sind für den Aufbau des Syntaxbaums zuständig und die `Semantic` Klassen implementieren mit Unterstützung weiterer Hilfsklassen die semantische Analyse der jeweiligen Sprachebene.

6.3.3 Codeanalyse

Durch die Bereitstellung der Klassendatenbank wird bereits ein breites Spektrum an Analyseaufgaben abgedeckt. Alle Resultate der semantischen Analyse sind dort zu finden. Wenn jedoch beispielsweise bestimmte syntaktische Konstrukte wie bestimmte Arten von Ausdrücken oder Schleifen manipuliert werden sollen, hilft die Klassendatenbank nicht weiter. In diesem Fall müssen die Syntaxbäume durchsucht werden, über die die Anwendung dann an die entsprechenden *Tokens* kommen kann.

Um solche Suchvorgänge im Syntaxbaum zu ermöglichen und gleichzeitig die Definitionen der verschiedenen Syntaxbaumknotenklassen von den Anwendungen fern zu halten, wurde der sogenannte *Match*-Mechanismus konzipiert. Er basiert auf der Idee des Mustervergleichs auf Baumstrukturen [56]. Dabei wird der Syntaxbaum eines Programms durchsucht, indem versucht wird, Unterbäume zu finden, die zu einem Suchbaum passen. Abbildung 6.6 auf der nächsten Seite illustriert diesen Vorgang. Sie zeigt links unten einen Suchbaum, der einen sogenannten *Any*-Knoten enthält. Ein solcher Knoten passt während der Suche zu jedem beliebigen Unterbaum.

Mit dem gezeigten Suchbaum können beliebige Wertzuweisungen an die Variable `x` gesucht werden. Der Syntaxbaum oben rechts repräsentiert ein Stück des durchsuchten Programms. Wie anhand der Markierung zu erkennen ist, passt das Suchmuster an einer Stelle. Der `x` zugewiesene Wert, für den im Suchmuster *Any* als Platzhalter angegeben wurde, entspricht bei der gefundenen Übereinstimmung einem *BinaryExpr*-Knoten für den Ausdruck `(47+11)*3`.

Ein solcher Mechanismus allein ist bereits sehr nützlich. Allerdings erfordert das Aufbauen von Suchbäumen genaue Kenntnisse über die entsprechenden Knotentypen. Diese Aufgabe kann jedoch ein erweiterter *Parser* der Anwendung abnehmen. Mit Hilfe des folgenden Codefragments kann der in Abbildung 6.6 dargestellte Suchbaum durch den *Parser* automatisch generiert werden:

```
\do-expression x = \any-expression
```

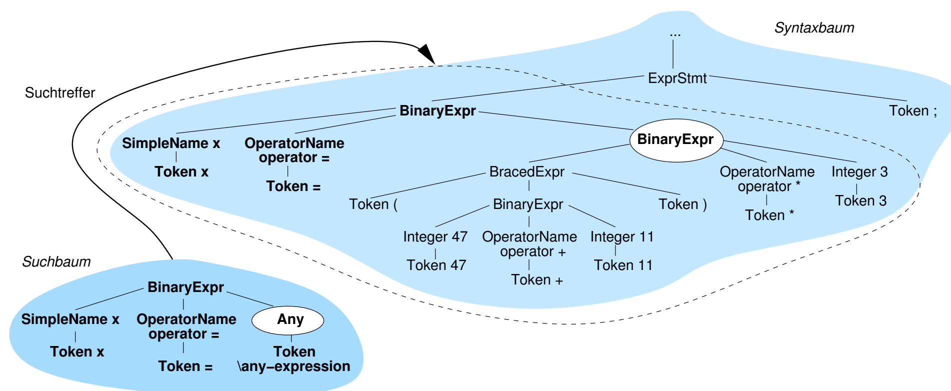


Abbildung 6.6: Der *Match*-Mechanismus von PUMA

Da ein C++ *Parser* im Normalfall versucht, ein komplettes Programm zu parsen, wird das Start-*Token* `\do-expression` benötigt. Es versetzt den *Parser* direkt in einen Zustand, in dem Ausdrücke akzeptiert werden. Die Grammatik muss dann lediglich um bestimmte Alternativen erweitert werden, so dass auch `\any-expression` als gültiger Ausdruck akzeptiert wird.

Durch diesen Mechanismus können flexible Suchoperationen auf dem Syntaxbaum durchgeführt werden, ohne dass die Anwendung Klassen wie *BinaryExpr* kennen muss. Für den Zugriff auf die korrespondierenden *Tokens* zwecks Manipulation genügt die Kenntnis einer gemeinsamen Basisklasse aller Syntaxbaumknotentypen.

6.3.4 Codemanipulation

Bei PUMA sind alle Codemanipulationen letztlich Manipulationen der *Token*-Ketten, die den verschiedenen *Units* zugeordnet sind. Dafür stehen Operationen wie Ausschneiden (*cut*), Löschen (*kill*), Kopieren (*copy*) und Einfügen (*paste*) zur Verfügung, wie sie auch die gängigen Textverarbeitungsprogramme bereitstellen. Der einzige Unterschied ist, dass bei PUMA auf Basis lexikalischer Einheiten und nicht auf Basis einzelner Zeichen manipuliert werden kann.

In Abbildung 6.4 auf Seite 74 wurde gezeigt, dass zwischen Syntaxbäumen und *Tokens* eine Verbindung besteht. Wenn nun durch die Anwendung beispielsweise *Tokens* gelöscht werden, würden diese Verbindungen ihre Gültigkeit verlieren. Ähnlich problematisch wäre die Situation, wenn temporär durch das Manipulieren syntaktisch ungültige oder auch neue Konstrukte entstünden. All diese Situationen machen es schwer, die Syntaxbäume und die Listen von *Tokens* konsistent zu halten. Dies ist jedoch für Suchvorgänge wichtig, bei denen über Syntaxbäume auf *Tokens* geschlossen wird.

Eine mögliche und bei PUMA verfolgte Lösung dieses Dilemmas besteht in der Anwendung eines Transaktionskonzepts. Die Idee ist in diesem Fall, dass die Ma-

nipulationsoperationen nicht direkt ausgeführt werden, sondern lediglich als auszuführende Operationen in einer Liste gespeichert werden. Damit bleiben die *Token*-Ketten und Syntaxbäume zunächst unberührt und konsistent, bis durch den Aufruf der `commit` Operation alle Änderungen durchgeführt werden. Im Idealfall benötigt eine Applikation nur eine solche Transaktion. Sollte dies nicht reichen, müssen die Syntaxbäume nach dem `commit` neu generiert werden, da sie nun nicht mehr zu den *Token*-Ketten passen.

6.3.5 Aspekte

Eine Besonderheit des Entwurfs von PUMA ist, dass Teile unter Berücksichtigung von Aspekten entstanden sind. So wurden drei *Crosscutting Concerns* im Bereich der *Parser*-Familie identifiziert:

Kontrollflussverfolgung: Um *Parser* während der Entwicklung und Wartung zu untersuchen, sind Ausgaben beim Betreten und Verlassen der Funktionen, die Regeln der Grammatik repräsentieren, hilfreich. Zudem ist es sinnvoll, das Einlesen von *Tokens* und das Akzeptieren oder Scheitern bei der Regelerkennung zu protokollieren. Der dafür nötige Code betrifft alle Funktionen der existierenden und zukünftigen `Syntax` Klassen. Durch eine aspektorientierte Realisierung werden die `Syntax` Klassen übersichtlicher. Darüber hinaus kann durch das statische Konfigurieren des Aspekts beispielsweise dafür gesorgt werden, dass im produktiven Einsatz der Bibliothek kein Programmcode zur Kontrollflussverfolgung enthalten ist.

Aufbauen von Suchbäumen: Im Zusammenhang mit dem *Match*-Mechanismus werden die Funktionen der `Syntax` Klassen, die Regeln der Grammatik repräsentieren, um spezielle Alternativen erweitert. Dadurch wird zum Beispiel statt eines Ausdrucks (*expression*) auch ein `\any-expression` *Token* im Quelltext akzeptiert. Durch entsprechende Erweiterungen in den `Builder` Klassen werden solche *Any-Tokens* im Syntaxbaum durch *Any*-Knoten repräsentiert. Dieser Mechanismus erlaubt das automatisierte Aufbauen von Suchbäumen aus Quelltextfragmenten, die mit speziellen *Any-Token* bestückt sind. Die Implementierung würde sich ohne Aspekte über viele Funktionen der verschiedenen `Syntax` und `Builder` Klassen erstrecken. Dabei existiert auch ein hohes Maß an Redundanz, so dass ein Aspekt hier die bessere Implementierungsform ist.

Sprachdialekte: Wenn Quelltexte zu analysieren sind, die zum Beispiel den GNU oder Microsoft Dialekt von C bzw. C++ benutzen, muss entsprechender Code zur Behandlung der Abweichungen oder Erweiterungen zum Sprachstandard im System vorhanden sein. Dieser Code betrifft die Syntax und Semantik sowie die Datentypen der Syntaxbaumknoten. Durch die saubere Trennung der Dialekte vom Standardcode wird die Wartung erleichtert.

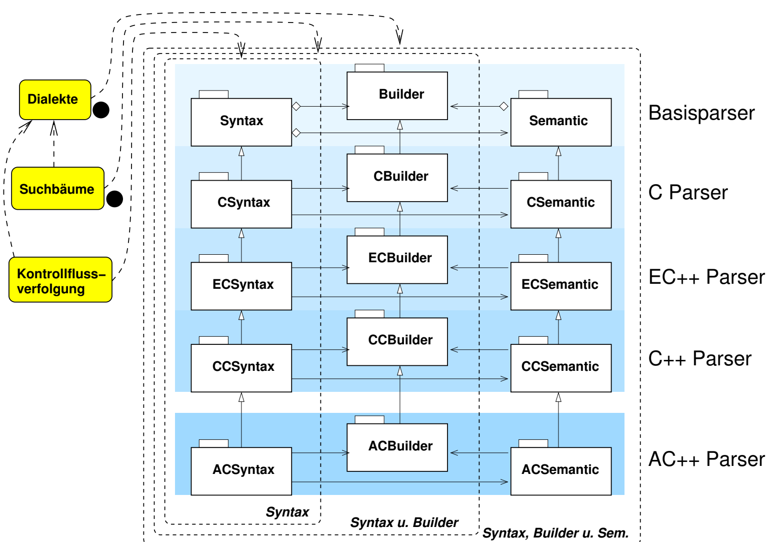


Abbildung 6.7: Aspekte des PUMA *Parsers*

Abbildung 6.7 zeigt die entsprechenden drei Aspekte und ihre Wirkung auf die *Parser*-Familie. Es ist zu erkennen, dass die Aspekte zum Aufbauen von Suchbäumen und die Sprachdialekte notwendige Einwirkungsbeziehungen haben. Diese Notwendigkeit hat keine zwingenden technischen Gründe. Es sollte jedoch vermieden werden, dass ein Aspekt Teil des Systems ist, obwohl er die Funktion, auf die sein Name schließen lässt, mangels passender Verbindungspunkte nicht erbringen kann.

Zwischen den Aspekten besteht eine Ordnungsbeziehung. So ist es wichtig, dass der Kontrollflussverfolgungsaspekt auch die Spracherweiterungen und Regeländerungen aufgrund von Dialekten berücksichtigt.

Da die notwendigen Voraussetzungen für das Verständnis der Implementierung dieser Aspekte erst im Laufe des folgenden Kapitels erläutert werden, soll an dieser Stelle nichts dazu gesagt werden. Stattdessen wird am Ende der Ausführungen zur Implementierungssprache AspectC++ als Benutzungsbeispiel für die Sprache und zur Vervollständigung des aktuellen Kapitels die Implementierung eines Sprachdialekts beschrieben.

6.4 Implementierung

Im Zuge der Vorbereitung dieser Dissertation wurde in den letzten Jahren PUMA entsprechend des Entwurfs, der im vorangegangenen Abschnitt beschrieben wurde, implementiert. Dabei entstanden etwa 70000 Zeilen[7] Programmcode. Die Tabelle 6.1 auf der nächsten Seite zeigt, wie sich diese auf die einzelnen PUMA

[7] gemessen mit dem UNIX Kommando `wc -l`

Subsysteme verteilen. Durch die beiden Zeilen für CParser (alt) und CParser (neu) wird eine Neuentwicklung des C++ *Parsers* dokumentiert. Der alte *Parser* wurde mit Hilfe eines *Parser*-Generators erstellt und der in Abschnitt 6.2.1 bereits angesprochene "lexical typedef hack" wurde verwendet, um Namen vor der Weitergabe an den generierten *Parser* näher zu beurteilen. Zu diesem Zweck wurden eigene Tabellen mit Symbolen und Gültigkeitsbereichen verwendet. Um den komplexen Regeln der Namenssuche (*lookup*) in C++ gerecht zu werden, entwickelte sich diese *Parser*-Vorstufe im Laufe der Zeit zu einem eigenen *Parser*, der bereits einiges an semantischer Analyse durchgeführt hat. Dadurch kam es zu einem hohen Maß an redundanten Aktionen und Datenstrukturen.

PUMA Subsystem	**Programmzeilen**	**Erläuterung**
Basics	4346	Grundlegende Hilfsklassen wie Container
CScanner	1771	Lexikalische Analyse
Compiler	5139	Grundlegende Klassen für Compiler *Front-Ends*, die nicht sprachspezifisch sind
CPreprocessor	5802	C/C++ Präprozessor
CParser (alt)	32079	Syntaktische und semantische Analyse
CParser (neu)	4449	Neuimplementierung des *Parsers* entsprechend Abschnitt 6.3.2
CClassDB	6311	Klassendatenbank
Manipulator	3781	Manipulationsinfrastruktur
Operations	6615	Bereits eingebaute Manipulationsoperationen

Tabelle 6.1: Umfang der PUMA Subsysteme

Die Neuentwicklung des *Parsers* beendet nun diesen Zustand. Hier wird entsprechend des Entwurfs in Abschnitt 6.3.2 die semantische Analyse konsequent gleichzeitig mit der syntaktischen Analyse durchgeführt. Zum Speichern der Informationen werden direkt die Strukturen der Klassendatenbank benutzt und für die Suche nach Namen werden keine gesonderten Datenstrukturen benötigt. Gleichzeitig sind syntaktische und semantische Analyse soweit wie bei C++ nur möglich getrennt. Obwohl die Implementierung des neuen *Parser*-Subsystems noch nicht abgeschlossen ist, deutet sich jetzt schon eine erhebliche Reduktion der Programmgröße und Verbesserung der Wart- und Erweiterbarkeit an.

Funktional deckt die PUMA Implementierung alle in Abschnitt 6.3 erläuterten Bereiche ab. Die *Parser*-Familie weist bisher K&R/ANSI/ISO C, ISO C++ und AspectC++ *Parser* auf. Die explizite Trennung eines EC++ *Parsers* ist bis-

her noch nicht erfolgt. Untersuchungen haben gezeigt, dass durch das Erben von *Parser*-Klassen ein hohes Maß an Wiederverwendung erzielt werden konnte. Dies verdeutlicht Abbildung 6.8. Die Funktionen der drei Klassen `Syntax`, `Builder` und `Semantic` wurden bereits in Abschnitt 6.3.2 beschrieben. Es ist zu erkennen, dass in allen drei Teilen des C++ *Parsers* jeweils nur ein Viertel der Regeln bzw. Funktionen völlig neu implementiert werden musste. Der Rest konnte entweder unverändert vom C *Parser* geerbt oder musste lediglich erweitert werden.

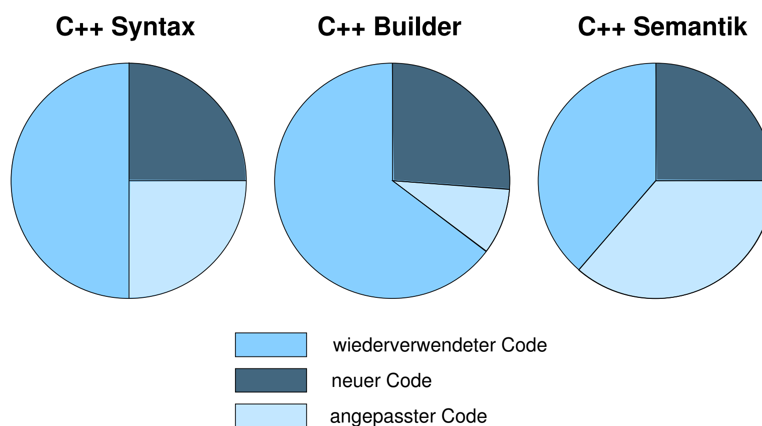


Abbildung 6.8: Wiederverwendung beim Übergang vom C zum C++ *Parser*

Der C *Parser* von PUMA deckt den vollständigen Umfang der Sprache bereits ab. Bei dem C++ *Parser* fehlen noch wenige aber dafür wichtige Sprachelemente: Namensräume und *Template*-Spezialisierung. Da beide jedoch bei der Implementierung von PURE nicht verwendet wurden, reicht der Sprachumfang der *Parser* für die in Kapitel 9 präsentierten Fallstudien aus.

Der in Abschnitt 6.3.3 beschriebene *Match*-Mechanismus wurde zu einem sehr flexiblen Analysewerkzeug ausgebaut. So kann auf der Ebene der Suchausdrücke das Suchverhalten beeinflusst werden und komplexe Bedingungen für Unterbäume von Suchtreffern können ebenfalls über *Match*-Ausdrücke und logische Verknüpfungen formuliert werden. Weitere Informationen dazu sind dem PUMA Benutzerhandbuch zu entnehmen [109].

6.5 Benutzung

6.5.1 Die Programmierschnittstelle

Die Programmierung von Codeanalyse- und Codemanipulationsoperationen erfolgt mit PUMA hautpsächlich auf Basis der Funktionen der Klassendatenbank. Als erster Schritt muss die Klassendatenbank jedoch mit Informationen gefüllt werden. Das folgende Codefragment zeigt, wie dies in einem C++ Anwendungsprogramm in der Regel aussieht:

```
PathIterator iter (".*\\.cc");
while (project.iterate (iter))
   classdb.Expand (iter.file ());
```

Die `Expand` Methode des `classdb` Objekts, das zuvor instanziiert werden muss, sorgt dafür, dass die als Argument übergebene Quelltextdatei syntaktisch analysiert wird und dass alle semantischen Informationen, die bei dieser Analyse anfallen, in den Strukturen der Klassendatenbank abgelegt werden. Die Schleife um diesen Aufruf herum sorgt dafür, dass dies für alle Dateien des Projekts mit der Dateiendung ".cc" durchgeführt wird.

Mit Hilfe einfacher Funktionen lassen sich nun Informationen über das Projekt abfragen. Hier wird zum Beispiel die Liste aller Klassen ausgelesen und der Name wird jeweils vollqualifiziert ausgegeben:

```
for (unsigned c = 0; c < classdb.ClassInfos (); c++)
   cout << classdb.ClassInfo (c)->QualName () << endl;
```

Über das von der Methode `ClassInfo` gelieferte Objekt kann die Anwendung in analoger Weise zum Beispiel Informationen über Attribute oder Methoden der jeweiligen Klasse erhalten.

Soll nun beispielsweise im Syntaxbaum einer bestimmten Methode eine Wertzuweisung an eine Variable gesucht werden, so kann dafür der *Match*-Mechanismus genutzt werden:

```
MatchCollector coll ("\\do-expression NAME = \\any-expression");
coll.defMacro ("NAME", name_der_variablen);
coll.collect (funktion->DefNode ());
for (unsigned m = 0; m < coll.Matches (); m++)
   commander.replace (coll.Match (m)->SubMatch ("root")->Tree (),
                      neuer_ausdruck);
commander.commit ();
```

Ein `MatchCollector` übersetzt den als Konstruktorparameter übergebenen *Match*-Ausdruck in einen *Match*-Syntaxbaum. Dabei ist es, wie in diesem Beispiel zu sehen ist, erlaubt, den Ausdruck durch Verwendung von Makros zu parametrisieren. Die Methode `collect` führt den eigentlichen Mustervergleich aus. Der dabei verwendete Parameter ist ein Verweis auf den Syntaxbaum der Funktion, in der gesucht werden soll. Er kann über die Klassendatenbank erfragt werden.

Die gefundenen Übereinstimmungen im Syntaxbaum können dann in einer Schleife abgefragt werden. In dem Beispiel wird nun für jeden Suchtreffer die Methode `replace` eines `ManipCommander` Objekts aufgerufen, wodurch die vom gefundenen Syntaxbaum referenzierten *Tokens* durch eine andere *Token*-Kette ersetzt

werden[8]. Durch den Aufruf der Methode `commit` wird dann die komplette Manipulationstransaktion durchgeführt.

6.5.2 Visuelles Entwickeln von Transformationsnetzwerken

Neben der direkten Programmierung auf Basis der PUMA Programmierschnittstelle wie sie im letzen Abschnitt gezeigt wurde, besteht noch eine ergänzende visuelle Möglichkeit, Analyse- und Transformationsoperationen zu implementieren. Diesem Ansatz liegt die Beobachtung zugrunde, dass die PUMA Operationen nur mit einer sehr kleinen Menge von Datentypen wie Repräsentanten für Klassen, Funktionen, Syntaxbäumen und *Tokens* umgehen und dass komplexe Operationen durch Zusammensetzen einfacher Operationen erstellt werden können. Eine grafische Umgebung vereinfacht diese Arbeit für den Programmierer und befreit ihn sogar von der Last, eine bestimmte Programmiersprache zu beherrschen.

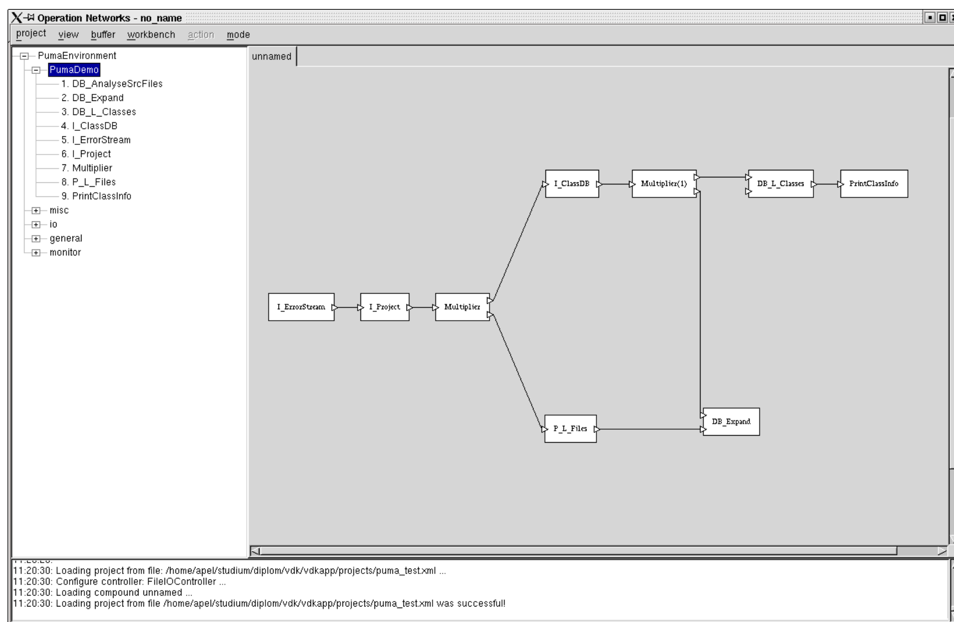


Abbildung 6.9: Das PUMA *Visual Development Kit*

Abbildung 6.9 zeigt die daraufhin entstandene Entwicklungsumgebung [5]. In dieser Umgebung werden die PUMA Operationen als Knoten von Datenflussnetzwerken betrachtet, die je nach Art des Knotens Informationen über ein Programm liefern, es manipulieren oder einfach nur Filter- oder Iterierungsaufgaben wahr-

[8] Objekte in unterschiedlichen Gültigkeitsbereichen können bei C oder C++ Programmen den selben Namen tragen. Wenn hier nur Manipulationen eines bestimmten Objekts gewünscht sind, kann mit Hilfe von Verbindungen zwischen dem Syntaxbaum und semantischen Informationen eine genaue Zuordnung ermittelt werden.

nehmen. Einmal zusammengestellte PUMA Operationsnetzwerke können gruppiert und als Einheit in Bibliotheken eingefügt werden.

Da diese Entwicklungsumgebung erst kürzlich fertiggestellt wurde, bestehen noch nicht sehr viele Erfahrungen mit der grafischen Entwicklung von PUMA Anwendungen oder Operationen. Der Ansatz ist jedoch vielversprechend, da er eine interaktive Entwicklung gestattet. So lassen sich zum Beispiel *Match*-Ausdrücke schrittweise entwickeln, ohne zwischendurch das Programm neu übersetzen zu müssen. Desweiteren erlaubt das Datenflussmodell in der Zukunft einen verhältnismäßig leichten Übergang zu einer vielfädigen Ausführung der PUMA Anwendungen, da Parallelität und Synchronisationspunkte explizit sichtbar sind.

6.6 Zusammenfassung

PUMA stellt eine leicht anzuwendende, flexible Basis für die Implementierung verschiedenster Aspektweber und beliebiger anderer Codeanalyse- und Codemanipulationsanwendungen dar. Durch die Bereitstellung eines erweiterbaren C++ *Parsers* inklusive semantischer Analyse wurde die im folgenden Kapitel vorgestellte Entwicklung von AspectC++ sehr stark vereinfacht. Darüber hinaus wurde PUMA für die Entwicklung weiterer Aspektweber eingesetzt, die in Kapitel 8 näher erläutert werden.

Im Vergleich zu anderen Codeanalyse- und Codemanipulationssystemen wie Sage++ [19] oder OpenC++ [22] zeichnet sich PUMA besonders durch die Qualität der semantischen Analyse, die leichte Erweiterbarkeit und die Möglichkeit aus, auch *Header*-Dateien manipulieren und als solche wieder ausgeben zu können. Nur dadurch ist es zum Beispiel möglich, Aspekte auch auf den Code von Bibliotheken wie PURE anzuwenden. Die leichte Erweiterbarkeit wird im folgenden Kapitel anhand der AspectC++ Grammatik demonstriert. Eine weitere Besonderheit von PUMA ist die aspektorientierte Implementierung, durch die Aspekte wie zu unterstützende Sprachdialekte von den eigentlichen Grammatiken getrennt werden. Dies macht sich positiv auf die Erweiterbarkeit, Wartbarkeit und Wiederverwendbarkeit bemerkbar.

Kapitel 7

AspectC++

In diesem Kapitel wird der Entwurf, die Implementierung und die Benutzung einer aspektorientierten Spracherweiterung für C++ namens AspectC++ beschrieben. Ergänzend zu den hier gemachten Ausführungen steht die entstandene Implementierung zum Herunterladen und Ausprobieren auf der Projektwebseite `http://aspectc.org/` zur Verfügung.

7.1 Anforderungen

Die Forschung im Bereich der aspektorientierten Softwareentwicklung ist noch vergleichsweise jung. Daher sind auch Erfahrungen, welche Sprachelemente am besten geeignet sind, um die Implementierung von *Crosscutting Concerns* zu modularisieren, bisher rar. Es zeichnet sich jedoch ab, dass es im Interesse der Programmierer nicht eine Vielzahl von *Concern*-spezifischen Aspektsprachen geben wird, sondern dass sich aspektorientierte Vielzweckspracherweiterungen wie AspectJ durchsetzen. Dies wird zum Beispiel daran deutlich, dass AspectJ aus den *Concern*-spezifischen Sprachen COOL und RIDL hervorgegangen ist [75].

Bei dem Entwurf einer Vielzweckaspektsprache, die für die Manipulation von C++ Programmen eingesetzt werden soll, bietet es sich an, Syntax und Semantik soweit wie möglich von C++ zu übernehmen. Auf diese Weise wird dem Programmierer der Einstieg erheblich erleichtert. Wenn es sich um eine echte Spracherweiterung handelt, können zudem der Aspekt- und Komponentencode mit demselben Übersetzer bearbeitet werden und dieselben Programmvariablen und dergleichen nutzen, was die Integration in bestehende Projekte vereinfacht.

Ob ein *Crosscutting Concern* mit Hilfe einer Aspektsprache modularisiert werden kann, hängt davon ab, welche Sprachelemente sie unterstützt. Einerseits soll die Sprache ausdruckskräftig und leicht zu erlernen sein, andererseits sollen möglichst viele verschiedene *Crosscutting Concerns* abgedeckt werden. Hier muss ein Kompromiss zwischen der Menge der gut unterstützten Anwendungsfälle und der Einfachheit und Schlagkräftigkeit der Sprachelemente gefunden werden. Da die Entwickler von AspectJ in dieser Frage wohl zur Zeit die meiste Erfahrung vorwei-

sen können, da C++ und Java sich recht ähnlich sind und da die Sprachelemente von AspectJ auch geeignet scheinen, um *Crosscutting Concerns* im Betriebssystemsektor auszudrücken, bietet es sich an, die Sprachkonzepte von AspectJ zu übernehmen. Nach einer Anpassung dieser Sprachkonzepte an die Eigenheiten der Sprache C++ besteht dann eine gute Ausgangsposition zum Sammeln von Erfahrungen, die zu einer Verfeinerung der Sprache oder Revidierung des Ansatzes führen können. In Analogie zu AspectJ wurde daher für die zu entwickelnde C++ Spracherweiterung der Name AspectC++ gewählt.

Durch diesen Ansatz ist außerdem gewährleistet, dass der Komponentencode, auf den Aspekte wirken, in gewöhnlichem C++ formuliert sein kann. Damit können die exisitierenden PURE Subsysteme unverändert weiterverwendet oder im Rahmen anderer Projekte ohne einwirkende Aspekte genutzt werden.

Die Anwendungsdomäne von AspectC++ soll im Kontext dieser Arbeit eine Betriebssystemfamilie sein. Neben der generell geförderten Wiederverwendbarkeit sind solche Familien sind besonders wertvoll, wenn bestimmte Ressourcen wie zum Beispiel Speicherplatz in so geringer Menge zur Verfügung stehen, dass eine Implementierung für ein offenes Anwendungsspektrum scheitern würde. Kleinste eingebettete Systeme, aber auch parallele Systeme, wo zur Vermeidung eines Flaschenhalses Kommunikationslatenzen minimiert werden müssen, sind gute Beispiele für sinnvolle Anwendungsbereiche von Betriebssystemfamilien. Wenn AspectC++ hier eingesetzt werden soll, ist dafür zu sorgen, dass der Ressourcenverbrauch zur Laufzeit der erzeugten Programme minimal ist. Ein Laufzeitsystem mit konstanter Größe muss daher vermieden werden. Stattdessen ist Code, der für das Erbringen einer bestimmten Leistung erforderlich ist, sozusagen "auf Anforderung" zu erzeugen. Daraus resultiert auch, dass ein mit AspectC++ übersetztes Programm, das lediglich den normalen Sprachumfang von C++ nutzt, nicht mehr Ressourcen verbrauchen darf, als wenn es direkt mit einem C++ Übersetzer übersetzt worden wäre. Aus demselben Grund soll auch kein Sprachkonstrukt ein aufwändiges Laufzeitsystem erforderlich machen.

AspectC++ Programme müssen auf diversen Zielplattformen laufen können. So wurde zum Beispiel PURE auf eine ganze Reihe von 8, 16, 32 und 64 Bit CPUs bzw. Mikrocontrollern portiert. Das soll sich durch den Einsatz aspektorientierter Entwurfs- und Implementierungstechniken natürlich nicht ändern müssen. Da nicht zu erwarten ist, dass Aspekte besondere Auswirkungen auf die Maschinencodegenerierung oder -optimierung haben, lohnt es daher auch nicht, Arbeit in die Implementierung solcher Funktionen zu stecken. Bei dem Entwurf von AspectC++ sollte vielmehr darauf geachtet werden, dass bestehende C++ Übersetzer möglichst weitgehend wiederverwendet werden können, sei es durch Anpassung des *Front Ends*[1] oder durch Transformation von AspectC++ Programmen in C++ Programme.

[1] Unter einem *Compiler Front End* versteht man die von der Zielsprache unabhängigen Teile des Übersetzers wie lexikalische, syntaktische und semantische Analyse.

7.2 Probleme

Die Probleme bei der Verfolgung des im vorangegangenen Abschnitts vorgeschlagenen Entwurfsansatzes, die Sprachkonzepte von AspectJ auf C++ zu übertragen, liegen im Bereich des Übersetzungvorgangs, der unterschiedlichen Syntax und Semantik von C++ und Java sowie in der bei Java vorhandenen Laufzeitunterstützung. Diese Problembereiche werden in den folgenden Abschnitten getrennt betrachtet.

7.2.1 Übersetzung

Der AspectJ Übersetzer `ajc` kennt die Trennung zwischen Dateien des eigenen Projekts und externen Dateien. Er bekommt bei seinem Aufruf die Liste aller Dateien des Projekts übergeben, die dann als Ganzes unter Berücksichtigung von Aspektdefinitionen in *Class Files* übersetzt werden. Damit können Aspekte, egal in welcher Quelltextdatei sie definiert wurden, auf Klassen in allen Dateien des Projekts wirken. Bei C++ ist das Schema, ein komplettes Projekt mit einem Aufruf zu übersetzen, nicht üblich. Hier werden normalerweise Übersetzungseinheiten einzeln betrachtet und die erzeugten Objektdateien werden am Ende durch den Binder verknüpft. Wenn bei AspectC++ dieses C++ Übersetzungsschema beibehalten werden soll, hat das zur Folge, dass beschrieben werden muss, welche Aspekte es gibt und ob sie auf die aktuelle Übersetzungseinheit wirken sollen. Anders als bei der Verwendung von extern definiertem Komponentencode, zum Beispiel Funktionen, kommt das ausdrückliche Einbinden einer Aspektbeschreibung per *Include*-Direktive dabei nicht in Frage, denn Aspekte sollen auf Übersetzungseinheiten wirken können, die ohne Kenntnis der Aspekte erstellt wurden.

Bei AspectJ ist es möglich, mit Hilfe von Vergleichsmustern Aspekte für viele Punkte des Systems gleichzeitig zu definieren, zum Beispiel eine Menge von Klassen um ein bestimmtes Attribut zu erweitern. Um dabei den Wirkungsbereich von Aspekten gezielt auf Teilbereiche des Projekts einzuschränken, besteht bei AspectJ die Möglichkeit, in Vergleichsmustern auf Basis von Paketnamen (engl. *packages*) zu filtern. Für AspectC++ ist ein entsprechender Mechanismus über Namensräume alleine nicht ausreichend, da diese zum einen von vielen Programmierern noch nicht konsequent eingesetzt werden und zum anderen auch die Definitionen aus *Header*-Dateien projektexterner Bibliotheken vor einer versehentlichen Manipulation geschützt werden müssen.

7.2.2 Syntax

Auf der Ebene der Syntax gibt es einige Konstrukte in AspectJ, die so nicht bei AspectC++ übernommen werden können. Das folgende Codefragment zeigt beispielsweise einen sogenannten *Pointcut Designator* in AspectJ:

```
call(FigureElement.set*(int,int))
```

Solche Ausdrücke beschreiben in AspectJ die Bedingung, bei deren Eintreten *Advice*-Code zu aktivieren ist. In diesem Beispiel ist die Bedingung das Erreichen eines Aufrufs einer Methode der Klasse `FigureElement`, deren Name mit `set` beginnt und die zwei Argumente vom Typ `int` aufweist. Mindestens der "." sollte hier durch "::" ersetzt werden, da dies die übliche Schreibweise für voll qualifizierte Namen in C++ ist. Auch das Zeichen "*", das als Platzhalter für beliebige Zeichenfolgen verwendet wird, muss im C++ Kontext ersetzt werden, da es sonst zur Mehrdeutigkeit wegen des zur Deklaration und Dereferenzierung von Zeigern benutzten "*"-Operators kommen würde. Abgesehen von diesen vergleichsweise einfachen Umstellungen erfordert die syntaktische Analyse solcher Vergleichsmuster für Funktionssignaturen eine erhebliche Erweiterung der C++ Syntax und auch die lexikalische Analyse wäre davon betroffen. So dürfte eine Zeichenfolge wie "do*" nicht grundsätzlich in die *Tokens* | Schlüsselwort `do` | Operator `*` | zerlegt werden. Stattdessen müsste im Kontext eines Vergleichsausdrucks | Namensmuster `do*` | geliefert werden. Es kommt damit zu einer Kontextabhängigkeit der lexikalischen Analyse.

Wegen der im Vergleich zu C++ einfach zu analysierenden Grammatik der Sprache Java scheinen solche Dinge die AspectJ Entwickler nicht vor ernsthafte Schwierigkeiten zu stellen. Bei der Erweiterung von C++ sollte jedoch versucht werden, die Syntaxerweiterungen so gering wie möglich ausfallen zu lassen und Kontextabhängigkeiten während der lexikalischen Analyse zu vermeiden.

Ein weiteres Problem bei der syntaktischen Analyse stellen die Einfügungen (*Introductions*) dar. Sie dienen in AspectJ der statischen Programmerweiterung. Mit ihrer Hilfe können Aspekte in zusätzliche Methoden und Attribute in Klassen einfügen. Diese Einfügungen sind in den Zielklassen quasi unsichtbar, das heißt, sie sind im Quelltext der Klassen nicht zu sehen, können aber vom Aspektcode benutzt werden. Im Fall der öffentlichen Einfügungen (*public introductions*) können sogar beliebige Klassen diese Einfügungen benutzen. Da in C++ bereits während der Syntaxanalyse Objekt- und Methodenzugriffe ausgewertet werden, bedeutet dies, dass auch schon während dieser Phase die Zielklassen von Einfügungen zu ermitteln sind. Nur so kann verhindert werden, dass Referenzierungen von "unsichtbaren" eingefügten Methoden oder Attributen zu Fehlern führen.

7.2.3 Semantik

Ein wichtiger Unterschied zwischen C++ und Java besteht im Zeigerkonzept. Während in C++ Zeiger auf Datenobjekte durch Typumwandlungen und Zeigerarithmetik beliebig manipuliert werden können und so Speicherzugriffe unter Annahme beliebiger Datentypen und Adressen gestattet sind, gibt es bei Java Zeiger nur indirekt in Form von Objektreferenzen. Dieser Unterschied wirkt sich auf die Semantik der zu entwerfenden Sprache AspectC++ aus, da AspectJ folgende Form von *Pointcut Designators* unterstützt:

```
set(FigureElement.x)
```

Durch diesen Ausdruck kann *Advice*-Code an alle schreibenden Zugriffe des Attributs `x` der Klasse `FigureElement` gebunden werden. In Java können solche Zugriffe leicht durch die syntaktische und semantische Analyse zur Übersetzungszeit bestimmt werden. In C++ besteht dagegen durch das Zeigerkonzept das sogenannte Alias-Problem. Das heißt, es können Zeigervariablen existieren, durch deren Wert das gleiche Attribut bei Dereferenzierung angesprochen werden kann. Solche Zugriffe zu erkennen ist ausgesprochen schwierig. Es gibt zwar zahlreiche theoretische Arbeiten zum Thema der Alias-Erkennung, doch gehen diese als Modell von einer Untermenge der Sprache C aus, so dass ein sicheres Erkennen von Aliasen zur Übersetzungszeit in einem realen Programm im Moment unmöglich zu sein scheint[2]. Helfen könnte nur ein Laufzeitsystem, das alle Zeigerzugriffe im gesamten Programm abfängt und die Adresse mit einer Liste relevanter Objektadressen vergleicht. Aufgrund des enormen zu erwartenden Ressourcenbedarfs wird diese Lösung jedoch nicht in Betracht gezogen. Stattdessen wird die Semantik solcher *Pointcut Designators* in AspectC++ gegenüber AspectJ abgeschwächt. Lediglich alle **namentlichen** Attributzugriffe wie `x`, `obj.x` und `ptr->x` führen zum Aufruf von *Advice*-Code. Andererseits sollte bei C++ der Zugriff auf Objekte über Aliase ohnehin vermieden werden, da dadurch das Attributschutzkonzept umgangen werden kann.

Ein weiterer Unterschied zwischen C++ und Java, der sich auf AspectC++ auswirkt, ist die Tatsache, dass in Java die Reihenfolge von Klassendefinitionen keine Rolle spielt. Dies gilt auch, wenn eine Klasse auf eine andere Bezug nimmt. Dagegen muss in C++ jeder Typ und jedes Objekt vor seiner Benutzung deklariert worden sein. Da dies im Falle zyklischer Beziehungen unmöglich ist, besteht die Möglichkeit, unvollständige Deklarationen vorab zu machen (*forward declarations*), so dass wenigstens der Name bereits bekannt ist. Zwar reichen zum Beispiel Vorabdeklarationen von Klassen aus, um Zeigervariablen auf Objekte dieser Klasse zu definieren, doch Methodenaufrufe, Attributzugriffe oder das Erben von der Klasse sind nicht möglich. Diese Problematik ist für das zu entwickelnde AspectC++ relevant, da der Entwickler von *Advice*-Code davon ausgeht, dass er auf die Klasse zugreifen kann, von der aus der *Advice*-Code aktiviert wurde. Gleichzeitig muss im Rahmen der Codegenerierung die *Advice*-Code-Aktivierung selbst implementiert werden. Wenn nun beispielsweise der *Advice*-Code durch Transformation zu einer Funktion wird und die Aktivierung zu einem Funktionsaufruf, könnte es sehr schnell zu dem beschriebenen Zyklus kommen und der C++ Übersetzer würde das generierte Programm zurückweisen. Dieses Problem muss im Rahmen einer geeigneteren Codegenerierung vermieden werden.

7.2.4 Laufzeitumgebung

Im Gegensatz zu C++ bietet Java einen Reflektionsmechanismus (*reflection*) [102]. Damit können Java und AspectJ Programme zur Laufzeit Informationen über das

[2] Einen guten Einstieg in die Thematik bietet die Dissertation von X. Zhang [118].

eigene Programm, wie zum Beispiel Informationen über Eigenschaften von Klassen und Methoden, erfragen. Im Falle von Methoden ist es sogar möglich, diese wie gewöhnliche Objekte zu benutzen und bei Bedarf über ein Objekt des Typs `Method` die zugeordnete Methode aufzurufen. Es ist zu prüfen, ob aspektorientierte Software einen solchen Reflektionsmechanismus grundsätzlich braucht oder ob man bei AspectC++ darauf verzichten kann. Wenn er gebraucht wird, müsste dafür ein Ersatz entwickelt werden.

AspectJ selbst greift im generierten Code auf die Java Laufzeitumgebung zu, um Objekttypen zu ermitteln. Außerdem werden durch eine Laufzeitumgebung, die zu AspectJ gehört, Informationen über gerade erreichte Verbindungspunkte bereitgestellt. Dabei werden die Datentypen der Reflektionsschnittstelle wie `Class` und `Method` benutzt. Mindestens für diese Zwecke muss bei Unterstützung der entsprechenden AspectJ Spracheigenschaften in AspectC++ Ersatz gefunden werden.

7.3 Entwurf

Dieser Abschnitt beschreibt den unter Berücksichtigung der Anforderungen und Probleme entstandenen Entwurf der Sprache AspectC++. Hierbei wird anhand der Grammatik zunächst ein grober Überblick über die Sprachelemente vermittelt, die dann in separaten Abschnitten genauer erläutert werden. Der Grundgedanke bei dem Entwurf war, die wesentlichen Konzepte von AspectJ zu übernehmen und an die Eigenschaften der Sprache C++ anzupassen. Dadurch kommt es natürlich auch zu Unterschieden zwischen AspectC++ und dem “Vorbild” AspectJ. Sie werden jeweils am Ende eines Abschnitts beschrieben und begründet.

7.3.1 Überblick

Abbildung 7.1 auf der nächsten Seite zeigt die notwendigen Regeln, um die Grammatik von C++ zu AspectC++ zu erweitern[3]. Im einzelnen wurden die Regeln für *class-head*, *declaration* und *member-declaration* erweitert und die restlichen vier in der Abbildung gezeigten Regeln wurden komplett neu für AspectC++ hinzugefügt. Damit ist der Erweiterungsaufwand an einem existierenden C++ *Parser* als sehr gering einzustufen. Eine Erweiterung der lexikalischen Analyse ist notwendig, um dort die in der Grammtik fett gedruckten Schlüsselworte `aspect`, `dominates`, `pointcut` und `advice` zu erkennen. Eine kontextabhängige Analyse, wie sie in Abschnitt 7.2.2 diskutiert wurde, ist dagegen nicht erforderlich. Die Einfachheit der Grammatik resultiert daraus, dass für die neuen Sprachelemente bestimmte Regeln der bestehenden Grammatik wiederverwendet werden konnten. Welche Auswirkungen dies für den Anwender hat, werden die folgenden Abschnitte anhand von Beispielen zeigen. Dort wird zunächst erklärt, wie mit Hilfe von *Pointcuts* und Vergleichsmustern entsprechend der Regeln für *pointcut-declaration* und *pointcut-expression* Mengen von Verbindungspunkten definiert werden. Der darauf folgende

[3] Die Regeln sind als Ergänzung zu der Grammatik von B. Stroustrup [101] zu lesen.

Abschnitt beschreibt, wie unter Beachtung der Regeln für *class-head*, *dominates-clause* und *advice-declaration Advice*-Code innerhalb von Aspekten definiert wird, der bei Erreichen eines bestimmten Verbindungspunktes aus einem *Pointcut* zu aktivieren ist. Andere wichtige Spracheigenschaften, die mit der Reflektionsfähigkeit von Aspektcode zu tun haben und sich nicht direkt in der Grammatik widerspiegeln, werden anschließend in Abschnitt 7.3.4 beschrieben.

---

class-head:
  `aspect` *identifier*$_{opt}$ *dominates-clause*$_{opt}$ *base-clause*$_{opt}$

dominates-clause:
  `dominates` *aspect-name*

declaration:
  pointcut-declaration
  advice-declaration

member-declaration:
  pointcut-declaration
  advice-declaration

pointcut-declaration:
  `pointcut` *declaration*

pointcut-expression:
  constant-expression

advice-declaration:
  `advice` *pointcut-expression* : *declaration*

Abbildung 7.1: Erweiterung der C++ Grammatik für AspectC++

7.3.2 *Pointcuts* und Vergleichsmuster

Pointcut-Begriff

Das wichtigste Sprachelement zur modularen Implementierung eines *Crosscutting Concerns* ist das Konzept der *Pointcuts*. Sie beschreiben eine Menge von Verbindungspunkten, indem sie festlegen, unter welcher Bedingung die Aktivierung eines Aspekts erfolgen soll. Dabei kann diese Bedingung zum Beispiel das Erreichen einer bestimmten Stelle im Code sein oder auch die Belegung einer Variablen mit einem bestimmten Wert. Wie an diesen Beispielen zu erkennen ist, können *Point-*

cuts teilweise zur Übersetzungszeit ausgewertet werden. In manchen Fällen können jedoch erst Überprüfungen zur Laufzeit Sicherheit über die Notwendigkeit einer Aspektaktivierung bringen. Diese dynamische Sichtweise wurde in AspectC++ von AspectJ übernommen.

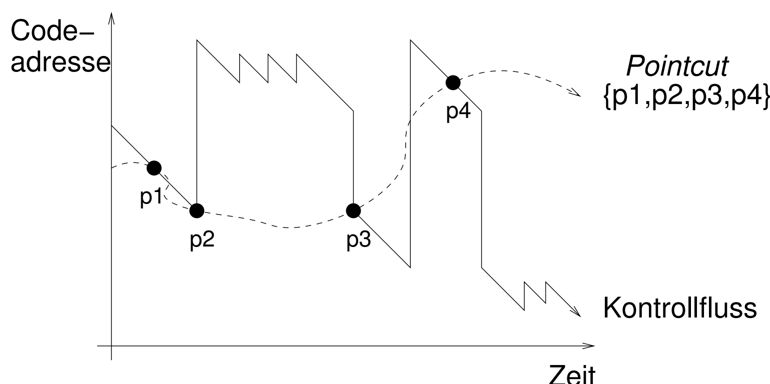


Abbildung 7.2: Schematische Erläuterung des *Pointcut*-Begriffs

Zur Erläuterung des Begriffs zeigt Abbildung 7.2 schematisch den Kontrollfluss eines Programms und den Schnitt durch bestimmte Programmpunkte, der durch einen *Pointcut* beschrieben wird. Auch in diesem Fall können die Verbindungspunkte nicht statisch an Codeadressen festgemacht werden. Das zeigt der Verbindungspunkt p4 auf der rechten Seite. Dieselbe Adresse wurde bereits zuvor durchlaufen, ohne dass es zu einer Aktivierung kam bzw. kommen sollte.

Vergleichsmuster

Das folgende Codefragment zeigt einen einfachen AspectC++ *Pointcut*-Ausdruck zur Beschreibung einer Menge von Verbindungspunkten. Es handelt sich dabei um alle Aufrufe von Funktionen der Klasse `C` deren Namen mit `init` beginnen:

```
call("void C::init%()")
```

Der als Zeichenkette angegebene Ausdruck ist ein sogenanntes Vergleichsmuster (*match expression*). Dadurch, dass solche Ausdrücke im Gegensatz zu AspectJ in Anführungszeichen gesetzt werden müssen, ist die syntaktische Analyse hier sehr einfach[4] und eine Kontextabhängigkeit der lexikalischen Analyse, wie sie in Abschnitt 7.2.2 diskutiert wurde, wird vermieden.

Vergleichsmuster für C++ Funktionssignaturen anzugeben, ist durch die komplexe Syntax sehr schwierig. Angenommen, es soll ein Muster für alle Funktionen mit einem Argument vom Typ `float` angegeben werden. Intuitiv wäre das Muster "% %(float)", wobei das erste "%" für den Rückgabewert und das zweite "%"

[4]Es kann die Regel *constant-expression* aus der C++ Grammatik wiederverwendet werden.

für den beliebigen Namen steht. Die folgenden Beispiele zeigen, dass der einfache textuelle Mustervergleich für viele Fälle aber scheitern würde:

```
void foo(float) // Übereinstimmung
int (*bar(float))(int) // Keine Übereinstimmung, falsch!
```

Die zweite Signatur führt zu keiner Übereinstimmung mit dem Muster, obwohl die beschriebene Funktion den Suchanforderungen entspricht. Es handelt sich um eine Funktion mit einem Argument von Typ `float` und einem Rückgabewert vom Typ "Zeiger auf eine Funktion mit `int` Argument und `int` Resultat". Bei einem einfachen textuellen Vergleich hätten Programmierer größte Schwierigkeiten, ihre Suchanforderungen in Suchmuster umzusetzen, daher wird vor dem Vergleich jede Signatur in folgendes Format umgewandelt:

```
<Resultatstyp> <Name>(<Argument>,<Argument>,...)
```

Damit würde die problematische Signatur in folgende Form transformiert werden:

```
int(*)(int) bar(float)
```

Nun würde der Mustervergleich gelingen. Hinzuzufügen ist noch, dass auch die Suchmuster in die Bestandteile `<Resultatstyp>`, `<Name>` und `<Argument>` zerlegt werden und dass der Vergleich der Signaturteile getrennt durchgeführt wird. Das "%" Zeichen kann entweder für einen ganzen Signaturteil stehen oder für einen Namen, das heißt eine Zeichenfolge aus Buchstaben und Ziffern. Drei Punkte ("...") in einer Argumentliste eines Vergleichsmusters passen zu beliebigen Argumentlisten. Tabelle 7.1 auf der nächsten Seite zeigt zusammenfassend einige Beispiele für Vergleichsmuster.

Arten von *Pointcuts*

AspectC++ kennt zwei Arten von *Pointcuts*. Die sogenannten **Code-*Pointcuts*** (*code pointcuts*) beschreiben einen Schnitt durch die Menge der Punkte im Kontrollfluss eines Programms wie es in Abbildung 7.2 auf der vorherigen Seite dargestellt wurde. Wie im Abschnitt 7.3.3 noch näher beschrieben werden wird, kann an Code-*Pointcuts* die Aktivierung von *Advice*-Code gebunden werden.

Die zweite Art von *Pointcuts* sind die **Namens-*Pointcuts*** (*name pointcuts*). Sie beschreiben einen Schnitt durch die Menge der in einem System bekannten Namen. Bei diesen Namen kann es sich im einzelnen um Typnamen, Klassennamen, Attributnamen, Funktionsnamen und Namen von Namensräumen handeln. Solche *Pointcuts* werden benötigt, um beispielsweise Klassen um Attribute oder Funktionen mit Hilfe der sogenannten Einfügungen (siehe 7.3.3) zu erweitern.

Pointcuts beider Arten können durch verschiedene *Pointcut*-Funktionen ineinander abgebildet werden. So ist beispielsweise ein Vergleichsmuster wie "% %(float)" bereits ein Namens-*Pointcut*. Es beschreibt die Menge aller Namen

Typvergleich	
int	der C++ Datentyp “int”
%*	Zeiger auf beliebige Klasse oder benannten C++ Datentyp
Namensraum-/Klassenvergleich	
Chain	die Klasse, Struktur, Union oder Namensraum “Chain”
Memory%	Klasse, Struktur, Union oder Namensraum, dessen/deren Name mit “Memory” beginnt
Attributvergleich	
Chain* Chain::next	ein Attribut der Klasse Chain namens next, das den Typ Chain* hat
% Chain::%	beliebiges Attribut innerhalb von Chain
Funktionsvergleich	
void reset()	die Funktion reset ohne Argumente und ohne Rückgabewert
% printf(...)	die Funktion printf mit beliebiger Argumentliste und Rückgabewert
void %(int,%)	eine Funktion mit zwei Argumenten und ohne Rückgabewert. Das erste Argument muss vom Typ int sein.

Tabelle 7.1: Beispiele für Vergleichsmuster

des Systems, die zu dem Muster passen. Durch die Anwendung der *Pointcut*-Funktion `call` wird daraus ein Code-*Pointcut* `call("% %(float)")`, der alle Punkte im Kontrollfluss beschreibt, an denen eine Funktion aus dem Argument-*Pointcut* aufgerufen wird. Neben Aufrufpunkten können Code-*Pointcuts* noch Ausführungspunkte sowie Setz- und Lesepunkte von Attributen enthalten. Code-*Pointcuts* können nur mit Hilfe von Namens-*Pointcuts* konstruiert werden, da alle genannten Arten von Verbindungspunkten nur durch die Nennung mindestens eines Namens beschreibbar sind.

Arten von Verbindungspunkten

Abbildung 7.3 zeigt die verschiedenen Arten von Verbindungspunkten und ihre Beziehungen anhand eines Beispiels. Code-Verbindungspunkte werden für die Bildung von Code-*Pointcuts* und Namens-Verbindungspunkte (oder kurz Namen) für Namens-*Pointcuts* eingesetzt. Jedem *Execution*-Verbindungspunkt ist genau ein Funktionsname zugeordnet. Es ist der Name der Funktion, auf deren Ausführung ein Aspekt reagieren können soll. Jedem *Call*-Verbindungspunkt sind zwei Funktionsnamen zugeordnet: der Name der Quell- und der Zielfunktion eines Aufrufs. Da innerhalb einer Funktion mehrere Funktionsaufrufe mit unterschiedlichem Ziel stattfinden können, wird einem Funktionsnamen eine ganze Liste von *Call*-Verbindungspunkten zugeordnet. Gleiches gilt für die *Get*- und *Set*-Verbindungspunkte, die eine Lese- bzw. Schreiboperation eines Attributs oder einer globalen Variablen beschreiben. Ihnen sind der Name der Funktion, in der sie sich befinden und der Name des benutzten Datenobjekts zugeordnet.

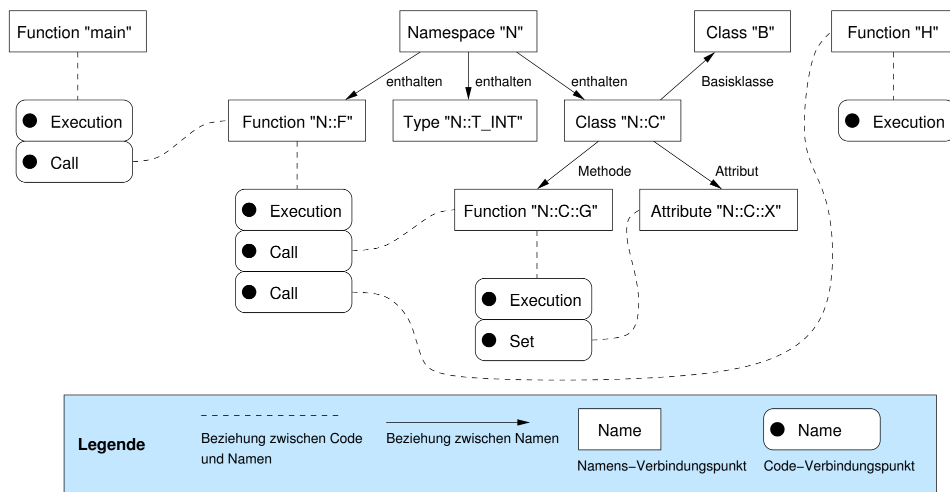


Abbildung 7.3: Verbindungspunkte und deren Beziehungen

Rein virtuelle Funktionen sind nicht ausführbar. Daher kann ihnen kein *Execution*-Verbindungspunkt zugeordnet sein. Dagegen ist der Aufruf einer solchen

Funktion, das heißt ein *Call*-Verbindungspunkt mit der rein virtuellen Funktion als Ziel, möglich.

Pointcut-Funktionen und -Ausdrücke

Mit call(<*Namens-Pointcut*>) wurde bereits eine sehr wichtige *Pointcut*-Funktion angesprochen. Solche Funktionen bilden eine Menge von Verbindungspunkten, das heißt einen *Pointcut*, in einen anderen *Pointcut* ab. Sie sind stark typisiert, das heißt Funktionen wie `call` lassen nur einen Namens-*Pointcut* als Argument zu. Code-*Pointcuts* werden zur Übersetzungszeit zurückgewiesen. Tabelle 7.2 auf der nächsten Seite zeigt eine vollständige Liste der *Pointcut*-Funktionen von AspectC++. Die Spalte “Typ” zeigt darin, welche Art von Argument-*Pointcut* bzw. *Pointcuts* erwartet wird und zu welcher Art von *Pointcut* das Resultat gehört. Dabei steht “N” für Namens-*Pointcut* und “C” für Code-*Pointcut*. Der in einigen Zeilen angegebene Index gibt Zusicherungen über die Art der Verbindungspunkte innerhalb des Resultats-*Pointcuts* an[5]. Auf der Argumentseite werden Verbindungspunkte, die für die jeweilige Funktion nicht relevant sind, ignoriert.

Ziel bei dem Entwurf der Tabelle war es, mit einer minimalen Zahl von Funktionen auszukommen und entlang der möglichen Beziehungen von Verbindungspunkten (siehe Abbildung 7.3 auf der vorherigen Seite) Abbildungen durchführen zu können. Dabei wurde bewusst davon abgesehen, wesentlich mehr Möglichkeiten bereitzustellen, als es durch AspectJ getan wird. Einige Erweiterungsmöglichkeiten liegen zwar auf der Hand, so beschränken sich Beziehungen zwischen Klassen nicht auf die Vererbungsbeziehungen, doch entsprechende *Pointcut*-Funktionen wie vielleicht `known`, `referenced` oder `aggregated` sollen erst hinzugefügt werden, wenn sich sinnvolle Anwendungsfälle ergeben.

Sehr wichtig sind die am Ende der Tabelle aufgeführten Mengenoperationen. Mit ihrer Hilfe können komplexe *Pointcut*-Ausdrücke aufgebaut werden, zum Beispiel:

```
(set("int N::C::X")||call("void N::C::G()"))&&!within("N::C")
```

Dieser Ausdruck liefert alle *Set*- und *Call*-Verbindungspunkte, die sich auf das Attribut `X` der Klasse `N::C` bzw. ihre Funktion `G()` beziehen und sich nicht innerhalb der Klasse selbst befinden.

Die Funktionen `that`, `target`, `args` und `result` liefern einen *Pointcut* mit Verbindungspunkten, die mit einer Bedingung verknüpft sind. In vielen Fällen kann diese Bedingung erst zur Laufzeit ausgewertet werden. Es geht dabei jeweils um die Frage, ob ein bestimmtes Objekt, das im Kontrollfluss gerade eine Rolle spielt, eine Instanz eines bestimmten Typs ist. Im Fall von `target` muss beispielsweise für das Zielobjekt eines Funktionsaufrufs eine Laufzeittypüberprüfung stattfinden,

[5] C, C_C, C_E, C_S, C_G: Code (beliebig, nur *Call*, nur *Execution*, nur *Set*, nur *Get*); N, N_N, N_C, N_F, N_T: Namen (beliebig, nur *Namespace*, nur *Class*, nur *Function*, nur *Type*)

Name	Typ	Resultat
call	N→C_C	Funktionsaufrufe
execution	N→C_E	Ausführung von Funktionen
set	N→C_S	Setzen von Attributen oder globalen Objekten
get	N→C_G	Lesen von Attributen oder globalen Objekten
within	N→C	alle Code-Verbindungspunkte innerhalb einer Funktion, Klasse, ...
that	N→C	Das aktuelle Objekt (`this`) ist eine Instanz einer bestimmten Klasse.
target	N→C	Das Zielobjekte (zum Beispiel bei Aufrufen) ist eine Instanz einer bestimmten Klasse.
args	(N,...)→C	Die Argumente sind Instanzen bestimmter Typen bzw. Klassen.
result	N→C	Das Resultat ist eine Instanz eines bestimmten Typs bzw. Klasse.
cflow	N→C	alle ausgeführten Code-Verbindungspunkte zwischen dem Eintritt in und dem Austritt aus bestimmten Funktionen
base	N→$N_{C,F}$	alle Basisklassen bzw. überdefinierte Funktionen aus Basisklassen
derived	N→$N_{C,F}$	alle abgeleiteten Klassen inklusive der Argumentklassen bzw. überdefinierende Funktionen aus abgeleiteten Klassen
&&	(N,N)→N, (C,C)→C	Schnittmenge
||	(N,N)→N, (C,C)→C	Vereinigungsmenge
!	N→N, C→C	Restmenge bzgl. aller Verbindungspunkte der entsprechenden Art des *Pointcuts*

Tabelle 7.2: Vordefinierte *Pointcut*-Funktionen

wenn die aufgerufene Funktion sich in einer Basisklasse der zu testenden Klasse befindet.

Eine weitere erst zur Laufzeit mit Sicherheit auswertbare Bedingung bringt die Funktion `cflow` mit sich. Sie beschreibt eine Menge von Kontrolflussbäumen. Ein bestimmter Verbindungspunkt gehört dabei nur zu dem Resultats-*Pointcut*, wenn er über einen Pfad aus der Menge dieser Bäume erreicht wurde.

Durch *Pointcut*-Deklarationen ist es möglich, *Pointcut*-Ausdrücke zu benennen und in verschiedenen Teilen eines Systems wiederzuverwenden. Bezüglich der Positionierung solcher Deklarationen im Quelltext sind keine speziellen Beschränkungen notwendig. Sie können global oder als Attribute von Klassen und Aspekten vorkommen. Es gelten dabei die normalen Sichtbarkeits- und Vererbungsregeln von C++, was im Zusammenhang mit wiederverwendbaren Aspekten in Abschnitt 7.3.3 interessant wird. Eine *Pointcut*-Deklaration sieht zum Beispiel wie folgt aus:

```
pointcut all_nodes() = derived("CTree");
```

Neben dem reinen Beschreiben von Verbindungspunkten kann ein *Pointcut* auch noch Variablen an Kontextinformationen des Verbindungspunktes binden. So können zum Beispiel aktuelle Argumentwerte von Funktionsaufrufen dem *Advice*-Code zugänglich gemacht werden. Die konkrete Umsetzung dieses Vorgangs in Form von Programmcode wird in Abschnitt 7.3.3 vorgestellt.

Unterschiede zu AspectJ

Der wesentliche Unterschied zwischen AspectJ und AspectC++ bezüglich der *Pointcuts* besteht darin, dass AspectC++ zwischen den *Pointcut* Arten Namens- und Code-*Pointcut* unterscheidet, während bei AspectJ Namen getrennt als sogenannte *Generalized Type Names* (GTN) behandelt werden. Auch mit GTNs können komplexe Ausdrücke gebildet werden:

```
Object+ && !java.io.*
```

Dieser GTN beschreibt alle Klassen, die von `Object` abgeleitet sind (der Operator '+' bildet die Menge aller angeleiteten Klassen), und deren Definition nicht aus dem Paket `java.io` stammt.

Auf den ersten Blick schienen die Konzepte der Namens-*Pointcuts* und der GTNs äquivalent zu sein. Dennoch hat die Variante von AspectC++ einige Vorteile (entsprechend der Reihenfolge ihrer Wichtigkeit):

1. GTNs in AspectJ können nicht benannt werden. Damit lassen sich solche Ausdrücke nicht wiederverwenden. Desweiteren können Namens-*Pointcuts* virtuell oder rein virtuell definiert werden. Das ist im Zusammenhang mit dem Erben von wiederverwendbaren, abstrakten Aspekten ein großer Vorteil. Nähere Erläuterungen zu diesen Konzepten folgen im nächsten Abschnitt.

2. Das Konzept der *Pointcut*-Funktionen, die auf Namen angewendet werden, wie `derived("Object")`, ist offener gegenüber Erweiterungen als die bei AspectJ gewählte Syntax wie `Object+`.

3. Die Grammatik von AspectC++ wird durch die Vereinheitlichung von Namen und Code deutlich vereinfacht.

Ein weiterer wichtiger Grund für diese Abweichung vom Vorbild liegt in der Sichtweise auf GTNs, die in der AspectJ Dokumentation eingenommen wird. Dort werden die Namen von Typen als statisch, das heißt über die Programmlaufzeit hinweg unveränderlich, angesehen. Da Code-Verbindungspunkte dagegen potentiell dynamisch sind, das heißt erst zur Laufzeit bestimmt werden können, ist die Trennung in GTNs und Code-Verbindungspunkten die logische Konsequenz. Die Annahme stimmt jedoch nur so lange wie ein System nicht zur Laufzeit rekonfiguriert wird. In einem solchen System müsste der Aspektweber beim Laden einer Klasse aktiv werden und überprüfen, ob darin Code-Verbindungspunkte sind oder ob die Klasse zum Beispiel durch Einfügungen (neue Attribute oder Methoden) erweitert werden muss. Somit werden auch Einfügungen zu Aktionen, die zur Laufzeit stattfinden müssen, doch das Ziel von Einfügungen wird bei AspectJ durch GTNs spezifiziert. Der Unterschied zu Code-*Pointcuts* und der Aktivierung von *Advice*-Code ist in diesem Szenario nur noch gering.

Natürlich ist eine Infrastruktur für die dynamische Rekonfigurierung von C++ Programmen schwer zu realisieren. Es sind im Moment auch keine konkreten Schritte geplant, um eine solche Infrastruktur durch geeignete Codegenerierung mit Hilfe von AspectC++ zu schaffen. Dennoch sollte der Weg dafür offen gehalten werden und lohnende Anwendungsfelder existieren. So ließen sich beispielsweise Wartungsarbeiten an laufenden Systemen durch das dynamische Laden von Aspekten elegant durchführen [34, 54].

7.3.3 Aspekte, *Advice* und Einfügungen

Advice-Code

Im vorangegangenen Abschnitt wurde beschrieben, wie Mengen von Verbindungspunkten in AspectC++ beschrieben werden können. Hier werden nun die Konzepte des *Advice*-Codes und der Einfügungen erläutert. Man kann beides als durch Aspekte ausgelöste Aktionen sehen, wobei die Einfügungen in der aktuellen Form der Sprache nur zur Übersetzungszeit ablaufen.

Advice-Code kann an einen Code-Verbindungspunkt gebunden werden und wird dann aktiviert, wenn dieser Punkt im laufenden System erreicht wird. Das folgende Codefragment zeigt eine entsprechende *Advice*-Deklaration:

```
advice execution("void disable_int()") : after () {
   cout << "Unterbrechungen sind gesperrt" << endl;
}
```

Jede *Advice*-Deklaration beginnt entsprechend der Grammatik (siehe Abbildung 7.1 auf Seite 91) mit dem Schlüsselwort `advice`. Gefolgt wird das Schlüsselwort von einem *Pointcut*-Ausdruck, der angibt, von wo aus und eventuell unter welcher Bedingung die Aktivierung des *Advice*-Codes stattfinden soll. Das `:after()` gibt an, dass die Codeaktivierung **nach** dem Erreichen des jeweiligen Verbindungspunktes stattfinden soll. Möglich sind hier auch `:before()`, das heißt **vor** dem Erreichen, oder `:around()`, wodurch der *Advice*-Code an Stelle des Verbindungspunktes ausgeführt wird. Der Name "around" – also drumherum – kommt daher, dass in diesem Fall der *Advice*-Code die Ausführung des Verbindungspunktes explizit anstoßen kann, so dass hier *Advice*-Code **vor und nach** dem Verbindungspunkt ablaufen kann.

Advice-Code und *Pointcuts* mit Kontextvariablen

Wie bereits in Abschnitt 7.3.2 angesprochen wurde, können mittels eines *Pointcut*-Ausdrucks auch Variablen des *Advice*-Codes an Kontextinformationen des aktuellen Verbindungspunktes gebunden werden. Diese Spracheigenschaft in Verbindung mit der Definition eines benannten *Pointcuts* zeigt das folgende Beispiel:

```
pointcut new_level(int l) =
   execution("void IRQ::level(...)") && args(l);

advice new_level(level) : after (int level) {
   cout << "Unterbrechungsebene " << level << endl;
}
```

Die erste Zeile definiert einen *Pointcut* namens `new_level` mit einer Kontextvariablen `l`. Damit wird zum Ausdruck gebracht, dass dieser *Pointcut* beim Erreichen der durch ihn beschriebenen Verbindungspunkte jeweils einen Wert vom Typ `int` liefert. In diesem Beispiel beschreibt der *Pointcut* den einen Punkt im System, bei dessen Erreichen die Unterbrechungsebene des Prozessors innerhalb eines Betriebssystems gesetzt wird. Die Kontextvariable liefert die Ebene, die dort gesetzt wird, da die Variable durch `args(l)` an das erste und einzige Argument der Funktion gebunden wird. Die Funktion `args` liefert hier einen *Pointcut*, der alle Verbindungspunkte des Systems enthält, bei denen ein Argument vom Typ `int` benutzt wird. Erst durch die Schnittmengenbildung mit `execution("void IRQ::level(...)")` wird genau der gesuchte Verbindungspunkt selektiert.

Die der *Pointcut*-Definition folgende *Advice*-Deklaration bindet nun die Ausführung von *Advice*-Code an das Erreichen eines Verbindungspunktes aus `new_level`. Die Kontextvariable zur Aufnahme des aktuellen Parameterwerts des erreichten Verbindungspunktes muss als formaler Parameter nach `after` bzw. `before` oder `around` deklariert werden. Diese Variable kann innerhalb des *Advice*-Codes wie ein gewöhnliches Funktionsargument benutzt werden.

Das Binden von Kontextvariablen erfolgt durch `that`, `target`, `args` und `result`. All diese *Pointcut*-Funktionen fungieren gleichzeitig als Filter entsprechend des

Typs der Kontextvariablen. So filtert in dem eben genannten Beispiel `args` alle Verbindungspunkte mit dem passenden Argumenttyp. Dieser Mechanismus erlaubt bereits den Zugriff auf sehr viele Kontextinformationen. Er erlaubt allerdings nicht die Implementierung von generischem *Advice*-Code, um beispielsweise von allen ausgeführten Funktionen die Argumente zu protokollieren, wenn diese unterschiedlich sind. Die für diese Zwecke nötigen Sprachelemente werden in Abschnitt 7.3.4 näher beschrieben.

Einfügungen

Einfügungen sind die zweite von AspectC++ unterstütze Form von *Advice*. Sie dienen der Erweiterung von Code und Datenstrukturen. Die folgenden Einfügungen erweitern beispielsweise zwei Klassen um je ein Attribut und eine Methode:

```
pointcut priobj() = "Thread" || "Bundle";

advice priobj() : int priority;
advice priobj() : void nice(int n) {
   priority += n;
}
```

In diesem Beispiel wird zunächst ein benannter Namens-*Pointcut* `priobj` definiert. Auch Einfügungen werden mit dem Schlüsselwort `advice` eingeleitet. Handelt es sich jedoch bei dem darauf folgenden *Pointcut* um einen Namens-*Pointcut*, so wird die nach dem ":" folgende gewöhnliche C++ Deklaration in die durch den *Pointcut* beschriebenen Klassen oder Namensräume eingefügt. Die erste Einfügung erweitert die Klassen `Thread` und `Bundle` um das Attribut `priority` und die zweite Einfügung erweitert die Klassen um die Methode `nice(int)`.

Einfügungen können zum einen die Implementierung von *Advice*-Code erleichtern, da bestimmte Funktionen, die auf Objektzustände zugreifen, in die entsprechenden Klassen verlagert werden können. Zum anderen können Einfügungen aber auch aus dem normalen Komponentencode heraus benutzt werden. So könnten beispielsweise im Sinne des *Subject-Oriented Programming* mehrere Aspekte durch Einfügungen Klassen und dementsprechend Objekte aus Fragmenten zusammensetzen, die dann von anderen Programmteilen als Einheit gesehen werden.

Darüberhinaus können Basisklasseneinfügungen dafür sorgen, dass eine Klasse unter Einwirkung des Aspekts eine bestimmte abstrakte Schnittstelle bereitstellt:

```
advice "State%" : baseclass(Persistant);
advice "State%" : void save (File f) { /* ... */ }
```

Hier wird beispielsweise in der ersten Zeile dafür gesorgt, dass alle Klassen, deren Name mit `State` beginnt, von der Klasse `Persistant` erben. Dabei soll es sich in diesem Beispiel um eine Schnittstellenklasse handeln, die eine Methode `save(File)` als rein virtuelle Funktion definiert. Die Einfügung in der zweiten Zeile sorgt dafür, dass alle `State`-Klassen eine konkrete Implementierung für `save(File)` enthalten.

Aspekte

Aspekte sind in AspectC++ ein Sprachkonstrukt, um Einfügungen und *Advice*-Code, die ein gemeinsames *Crosscutting Concern* implementieren, in modularer Weise zusammenzufassen. Dadurch werden sie in die Lage versetzt, gemeinsame Zustandsinformationen zu verwalten. Außerdem können sie durch herkömmliche Methoden unterstützt werden. Dementsprechend sehen Aspektdefinitionen wie erweiterte Klassendefinitionen aus:

```
aspect Counting {
   pointcut counted() = "Thread" || "Bundle";
   static int count;
   Counting () : count(0) {}
   advice counted() : class Helper {
                         Helper () { Counting::count++; }
                      } counter;
   advice execution("% main(...)"): after () {
      cout << "gezählt: " << count << " Objekte" << endl;
};
```

In diesem Beispiel wird die Anzahl der Objektinstanziierungen für eine Menge von Klassen bestimmt. Dazu wird durch eine Einfügung ein Attribut hinzugefügt, dessen Konstruktor einen globalen Zähler des Aspekts inkrementiert. Durch *Advice*-Code für die Funktion `main()` wird dafür gesorgt, dass bei Beendigung des Programms das Ergebnis der Zählung ausgegeben wird. Dieses Beispiel lässt sich leicht ausbauen, um beispielsweise festzustellen, ob die Zahl der Objektinstanziierungen gleich der Zahl der Zerstörungen für eine Gruppe von Klassen ist.

Um Wiederverwendung von Aspekten zu erlauben, wurden im Kontext von AspectJ abstrakte Aspekte konzipiert, von denen konkrete Aspekte erben können. Dieses Konzept wurde in AspectC++ übernommen, so dass der im letzten Beispiel gezeigte Aspekt wie folgt abstrakt definiert werden könnte:

```
aspect Counting {
   pointcut counted() = 0; // Ein rein virtueller Pointcut
   static int count;
   // ... entsprechend dem letzten Beispiel
};
```

Dieser Aspekt könnte in einer Bibliothek wiederverwendbarer Aspekte abgelegt und wie folgt in einem konkreten Projekt benutzt werden:

```
aspect ThreadCounting : public Counting {
   pointcut counted() = "Thread"||"Bundle";
};
```

Durch diesen Vererbungsmechanismus kann das “wie” vom “wo” eines Aspekts getrennt werden.

Aspekte werden standardmäßig automatisch als globale Objekte instanziiert. Die Idee ist dabei, dass Aspekte globale Systemeigenschaften zur Verfügung stellen und damit im Regelfall immer vorhanden und von überall zugänglich sein müssen. In einigen Sonderfällen bietet es sich jedoch an, von dieser Standardinstanziierungsform abzuweichen. Denkbar wäre beispielsweise im Betriebssystemkontext das Instanziieren eines Aspekts pro Faden oder Prozess. Der im letzten Beispiel betrachtete Aspekt `ThreadCounting` könnte dann die Anzahl der Fadenerzeugungen pro Faden, das heißt die Zahl der Kindfäden, bestimmen. Zu diesem Zweck müsste der Aspekt die sonst generierte statische Funktion `aspectOf` definieren, die jederzeit eine Aspektinstanz liefern kann. Dies zeigt das folgende Codefragment:

```
aspect ThreadCounting : public Counting {
   pointcut counted() = "Thread" || "Bundle";
   advice "Actor" : ThreadCounting instance;
   static ThreadCounting *aspectOf () {
      return life->instance;
   }
};
```

In dem diesem Beispiel zugrunde liegenden hypothetischen System, das an PURE angelehnt ist, verweist die globale Variable `life` auf das gerade aktive Fadenobjekt vom Typ `Actor`. Durch die Einfügung in die Klasse `Actor` wird dafür gesorgt, dass zu jedem aktiven Fadenobjekt eine Aspektinstanz existiert. Die Funktion `aspectOf` erlaubt jederzeit den Zugriff auf diese Instanz, was für *Advice*-Code und die darin vorkommenden Attributzugriffe notwendig ist.

Leider würde der jetzt modifizierte Aspekt `ThreadCounting` trotz `aspectOf` Funktion nicht die Kindfäden, sondern immer noch die Summe aller Fadeninstanziierungen zählen. Der Grund ist die Verwendung einer statischen Variablen für den Zähler `count` im Aspekt `Counting`. Um diese Schwäche zu beseitigen, müsste `count` in ein nicht-statisches Attribut umgewandelt werden. Dann stellt sich jedoch das Problem, dass der eingefügte Code der Hilfsklasse `Helper` an die richtige Aspektinstanz kommen müsste. Zwar wäre prinzipiell ein Aufruf der Funktion `aspectOf` denkbar, doch der Aspekt `Counting`, zu dem der eingefügte Code gehört, kennt die Methoden der von ihm abgeleiteten Aspekte nicht. Die hier nötigen Informationen und deren sprachliche Einbindung werden in Abschnitt 7.3.4 vorgestellt.

Unterschiede zu AspectJ

Die drei Konzepte Aspekt, *Advice* und Einfügungen von AspectC++ stimmen im Wesentlichen mit denen von AspectJ überein. Der Hauptunterschied besteht in der Aspektinstanziierung. Auch in AspectJ sind Aspekte standardmäßig globale Objekte. Für nicht globale Instanziierungen werden allerdings die folgenden Varianten von Aspektdefinitionen benutzt:

```
aspect <Name> perthis(<Pointcut>) {...}
aspect <Name> pertarget(<Pointcut>) {...}
```

```
aspect <Name> percflow(<Pointcut>) {...}
aspect <Name> percflowbelow(<Pointcut>) {...}
```

Damit werden die Instanziierungen der Aspekte von *Pointcuts* abhängig gemacht, die den AspectC++ Code-*Pointcuts* entsprechen. Die Instanziierung mit `perthis` bedeutet zum Beispiel, dass eine Aspektinstanz für jedes gerade bearbeitete Objekt (`this`) an den Verbindungspunkten des *Pointcuts* erzeugt wird. Der statische Methodenaufruf `<Name>.aspectOf(<Objekt>)` liefert den einem Objekt zugeordneten Aspekt.

Der Nachteil dieser Instanziierungsform ist, das Code-*Pointcuts* erst zur Laufzeit endgültig bestimmt werden können. Daher werden die Aspekte erst instanziiert, wenn ein Verbindungspunkt aus dem *Pointcut* das erste Mal erreicht wird. Dies führt zu einem größeren Aufwand zur Laufzeit als das auf Einfügungen basierende Schema bei AspectC++. Zudem bestünde bei C++ das Problem, den Speicher für die Aspektinstanzen irgendwann wieder explizit freigeben zu müssen. In AspectJ übernimmt der *Garbage Collector* diese Aufgabe. Dazu kommt, dass eine Instanziierung pro Faden, wie sie in den vorangegangenen Beispielen gezeigt wurde, mit dem Schema von AspectJ nicht möglich wäre, da dieses das Betreten einer Klasse voraussetzt. Insofern ist die Lösung von AspectC++ flexibler.

Ein weiterer Unterschied besteht im Schutzkonzept. Bei AspectJ dürfen Aspekte normalerweise nicht auf private oder geschützte Attribute oder Methoden anderer Klassen zugreifen. Da solche Zugriffe jedoch bei manchen Aspekten notwendig sind, kann ein Aspekt auch als “privilegiert” deklariert werden:

```
privileged aspect <Name> {...}
```

Bei AspectC++ wird dagegen ein anderes Schema benutzt. Hier hat *Advice*-Code grundsätzlich keine besonderen Zugriffsrechte, während Einfügungen wie Elemente der Zielklasse volle Zugriffsrechte besitzen. Damit werden kritische Zugriffe in eingefügte Funktionen ausgelagert und der Entwickler sollte anstreben, mit einem Minimum an Einfügungen auszukommen.

Beide Ansätze haben ihre Vor- und Nachteile. So können in AspectJ alle Teile eines Aspekts privilegierte Zugriffe durchführen, auch wenn nur ein einziger solcher Zugriff nötig sein sollte. Bei AspectC++ hat dagegen jede Einfügung die vollen Rechte, auch wenn diese im Einzelfall nicht nötig sind. Daher kann man sich in dieser Frage von der einfacheren Implementierung leiten lassen, die auf der Seite der AspectC++ Variante liegt.

7.3.4 Laufzeitunterstützung

Unterstützung für *Advice*-Code

Wie bereits angesprochen wurde, reicht für viele Aspekte der Mechanismus der Kontextvariablen nicht aus, um genügend Informationen über den Verbindungspunkt, von dem aus *Advice*-Code aktiviert wurde, zu erlangen. So müsste ein

Kontrollflussverfolgungsaspekt für eine vollständige Protokollierung von Funktionsaufrufen in einem System zur Laufzeit Informationen über Funktionsargumente und deren Typen haben, um Ausgaben typgerecht aufbereiten zu können. Zu diesem Zweck steht in *Advice*-Code und in Code von Einfügungen die Klasse `JoinPoint` zur Verfügung, über deren Methoden entsprechende Information über den aktuellen Verbindungspunkt zu bekommen sind.

statische Methoden:	
int args()	Anzahl der Argumente
AC::Type type()	Typ der Funktion/des Attributs
AC::Type argtype(int)	Argumenttypen
const char *signature()	Signatur der Funktion/desAttributs
unsigned id()	Kennung des Verbindungspunktes
AC::Type resulttype()	Resultatstyp
AC::JPType jptype()	Art des Verbindungspunktes
nicht-statische Methoden:	
void *arg(int)	Aktuelle Parameter
Result *result()	Resultatswert
That *that()	von `this` referenziertes Objekt
Target *target()	Zielobjekt eines Aufrufs
void proceed()	Bearbeitung fortführen
AC::Action &action()	*Action*-Struktur (siehe Text)
Typen:	
AC::Type	Objekttypkodierung nach C++ ABI
AC::JPType	Arten von Verbindungspunkten
Result	jeweiliger Resultatstyp
That	jeweiliger Objekttyp
Target	jeweiliger Zielobjekttyp

Tabelle 7.3: Schnittstelle der Klasse `JoinPoint` für *Advice*-Code

Tabelle 7.3 beschreibt die verschiedenen Methoden und Datentypen, die innerhalb von *Advice*-Code über die Klasse `JoinPoint` benutzt werden können. Dabei werden durch Typen und statische Methoden die Informationen, die bei jeder *Advice*-Code-Aktivierung gleich sind, geliefert. Nicht-statische Methoden liefern die Informationen, die sich bei jeder Aktivierung unterscheiden können. Als Objektname wird in diesem Fall `thisJoinPoint` verwendet.

Das Codefragment in Abbildung 7.4 auf der nächsten Seite zeigt, wie mit Hilfe dieser API ein wiederverwendbarer Kontrollflussverfolgungsaspekt implementiert werden kann. Dieser Aspekt sorgt dafür, dass jede Ausführung einer Methode, die durch einen virtuellen *Pointcut* erst in einem abgeleiteten Aspekt definiert wird, protokolliert wird. Die Hilfsfunktion `printvalue` sorgt für die typgerechte Aufbereitung der Ausgabe. An dieser Stelle wurde bewusst darauf verzichtet, einen ge-

```
#include "mangle.h"

aspect Trace {
  pointcut virtual methods() = 0; // relevante Methoden
  advice execution(methods()) : around() {
    cout << “before ” << JoinPoint::signature() << "(";
    for (unsigned i = 0; i < JoinPoint::args(); i++)
      printvalue(thisJoinPoint->arg(i),
                 JoinPoint::argtype(i));
    cout << “)” << endl;
    thisJoinPoint->proceed();
    cout << “after” << endl;
} };
```

Abbildung 7.4: Ein wiederverwendbarer Kontrollflussverfolgungsaspekt

nerischen Typ für Argumente bereitzustellen, da die Ansprüche an einen solchen Typ von Anwendung zu Anwendung stark differieren könnten. Stattdessen wird eine Minimallösung verfolgt, die bedeutet, dass lediglich ein Zeiger auf das Datenobjekt und ein Zeiger auf eine sehr kompakte Kodierung des Datentyps auf Anforderung geliefert wird. Mit Hilfe dieser Informationen und außerhalb der Sprache angesiedelter Hilfsfunktionen oder -klassen kann jede Anwendung den gewünschten Komfort beim Umgang mit generischen Datenobjekten bekommen.

Die Struktur `AC::Action` stellt einen ersten Ansatz dar, das Konzept der **Aktionen** umzusetzen. Damit ist in diesem Zusammenhang die Anweisungsfolge gemeint, die einem erreichten Verbindungspunkt im laufenden System folgen würde, wenn nicht *Advice*-Code aktiviert worden wäre. Das Fortführen des Kontrollflusses durch *Advice*-Code mit Hilfe von `thisJoinPoint->proceed()` ist demnach das Ausführen der Aktion des Verbindungspunktes. Bei dem bisherigen Modell der Verbindungspunkte von AspectC++ kann es sich dabei um Methodenaufrufe und -ausführungen sowie das Schreiben oder Lesen von Attributen handeln. Durch die Bereitstellung solcher Aktionen als Objekte, die alle für die Ausführung notwendigen Kontextinformationen, wie Parameter für Funktionsaufrufe, speichern, könnten Aktionen gespeichert und mittels `<Action-Objekt>.trigger()` verzögert ausgeführt werden. So ergeben sich interessante Anwendungen wie wiederverwendbare Aspekte, die die asynchrone Bearbeitung von Aufgaben veranlassen und verwalten, Aspekte, die ein transaktionsartiges Verhalten von Objekten verursachen, oder auch generische Pufferaspekte für die Resultate von idempotenten Funktionen. Leider fehlt noch Konzeptarbeit, um all diese Anwendungsfälle möglich zu machen. Weitere Erläuterungen dazu folgen in Abschnitt 7.3.5.

Unterstützung für Einfügungen

Code, der durch Einfügungen in fremde Klassen, Namensräume oder dergleichen gerät, kann ebenfalls über eine Klasse namens `JoinPoint` Informationen über den Verbindungspunkt erlangen. Tabelle 7.4 zeigt die für diese Art von Verbindungspunkten verwendbare Schnittstelle.

statische Methoden:	
const char *signature()	Signatur des Verbindungspunktes (zum Beispiel Klassenname)
unsigned id()	Kennung des Verbindungspunktes
AC::JPType jptype()	Art des Verbindungspunktes
Typen:	
Aspect	Typ des für die Instanziierungen verantwortlichen Aspekts

Tabelle 7.4: Schnittstelle der Klasse `JoinPoint` bei Einfügungen

Das folgende Codefragment zeigt eine nützliche Anwendung dieser Schnittstelle:

```
aspect TypeInfo {
  pointcut typed() = 0;
  advice typed() : static unsigned type_id() {
    return JoinPoint::id();
  }
  advice typed() : virtual unsigned type() { return type_id(); }
};
```

Durch diese zwei Einfügungen erhält eine Menge von Klassen eine statische und eine nicht-statische Methode zur Bestimmung einer Typkennung, die wie folgt angewendet werden kann:

```
if (obj->type() == MyClass::type_id()) ...;
```

Durch diesen Mechanismus werden bestimmten Klassen und deren Instanzen Typinformationen zugeordnet, die zur Laufzeit abgefragt werden können. Wenn der Übersetzer das Abschalten des C++ RTTI Mechanismus unterstützt, ist diese Implementierung in der Regel ressourcensparender.

Die Definition des Typs `Aspect` ist wichtig für den Zugriff auf Attribute und Methoden des Aspekts durch eingefügten Code. Dazu kann `Aspect::aspectOf()` aufgerufen werden. Diese statische Funktion existiert immer, da sie generiert wird, falls sie in der Aspektdefinition nicht explizit definiert wurde. Sie liefert einen Zeiger auf die jeweilige Aspektinstanz. Unter Anwendung dieser Technik lässt sich auch das in Abschnitt 7.3.3 begonnene Beispiel eines Kindfadenzählers vervollständigen:

```
int count;
void inc() { count++; }
advice counted() : class Helper {
  Helper () { Aspect::aspectOf()->inc(); }
} counter;
```

Unterschiede zu AspectJ

In AspectJ existiert wie in AspectC++ ein Objekt namens `thisJoinPoint`, über das *Advice*-Code Informationen über den aktuellen Verbindungspunkt erlangen kann. Die so bereitgestellte Schnittstelle schließt sich nahtlos an den Java Reflektionsmechanismus an, in dem Java Datentypen wie `Class`, `Method` und `Object` verwendet werden. Ein `thisJoinPoint`-Objekt für Einfügungen existiert in AspectJ nicht. Dafür bietet der Java Reflektionsmechanismus allerdings die Möglichkeit, Informationen über die Zielklasse einer Einfügung im eingefügten Code zu erlangen.

Bei C++ existieren solch detaillierte Informationen zur Laufzeit nicht. Alle Mechanismen, die in diese Richtung gehen und von Aspekten benötigt werden, müssen daher durch Erweiterungen von AspectC++ zur Verfügung gestellt werden. Hier wurde ein minimalistischer Ansatz verfolgt, der es einerseits gestattet, generischen Aspektcode zu formulieren und gleichzeitig keine fixen Ressourcenansprüche stellt. Erweiterungen der `JoinPoint` Schnittstelle sind möglich, obwohl anwendungsspezifische Erweiterungen außerhalb der Sprache favorisiert werden.

7.3.5 Geplante Erweiterung

Viele Erweiterungen an den bis hier vorgestellten Konzepten der Sprache AspectC++ wurden bereits überdacht. So fehlt bisher eine Möglichkeit, um über die `JoinPoint` Schnittstelle bei Einfügungen an Attribute und deren Typinformationen zu gelangen. Damit könnten Anwendungen realisiert werden, bei denen es notwendig ist, Objektinhalte über Adressraumgrenzen zu übertragen oder zu speichern.

Desweiteren muss noch an dem Konzept der Aktionen gearbeitet werden. Die Schwierigkeit dabei ist, dass die für die verzögerte Ausführung der Aktion notwendigen Kontextinformationen gespeichert werden müssen, aber nicht bekannt ist, welche Informationen das im Einzelnen sind. Wenn beispielsweise eine Aktion in der Ausführung einer Suche im Baum besteht, müsste unter Umständen der komplette Baum gesichert werden. Ob dies gewünscht ist, dürfte von Fall zu Fall unterschiedlich sein. Daher sollte man hier keine Pauschallösung anstreben. Das Problem ist vergleichbar mit der Situation beim Übertragen von Argumenten von Prozedurfernaufrufen (RPCs). Dort werden häufig Annotationen verwendet, um Stumpfgeneratoren anwendungsspezifische Hinweise zu geben [86]. Im Kontext von AspectC++ geht ein erster Ansatz in eine andere Richtung. Er besteht in der Idee, die Schnittstelle, die für das Speichern der Kontextinformationen in `Action` Objekten benutzt wird, zu dokumentieren, so dass der Anwendungscode über einen Aspekt und *Advice*-Code auf das Verhalten dieser Funktion Einfluss nehmen kann.

7.4 Implementierung

7.4.1 Struktur

Beim Entwurf der AspectC++ Grammatik wurde speziell darauf geachtet, die Erweiterungen gegenüber C++ möglichst einfach zu halten, damit auf bestehende *Parser* zurückgegriffen werden könnte. Allerdings wurde von der Erweiterung eines existierenden C++ Übersetzers abgesehen, da heutige Übersetzer nicht dazu ausgelegt sind und es auch keinen gibt, der Code für alle Zielplattformen der PURE Betriebssystemfamilie, dem ersten Anwendungsgebiet von AspectC++, erzeugt. Stattdessen wurde der Weg verfolgt, mit dem C++ seinerzeit als Erweiterung für C eingeführt wurde. Damals wurde mit einem Programm namens `cfront` [6] ein C++ Quelltext in einen C Quelltext transformiert, der dann mit einem beliebigen C Übersetzer weiterverarbeitet werden konnte. Analog dazu sollten AspectC++ Programme also mit Hilfe einer Quellcodetransformation in C++ Programme überführt werden.

Die Implementierung basiert auf den PUMA Analyse- und Transformationsfunktionen für C++ Code. Wie bereits in Kapitel 6 skizziert wurde, sind dazu verschiedene PUMA Klassen erweitert worden. Abbildung 7.5 auf der nächsten Seite zeigt beispielsweise einen Ausschnitt aus der Implementierung der Klasse `ACSyntax`. Sie stellt die AspectC++ Syntaxanalyse entsprechend der Grammatik in Abbildung 7.1 auf Seite 91 zur Verfügung. Dazu erbt sie von der PUMA Klasse `CCSyntax` die Regeln der C++ Syntax und definiert unter anderem die Regel `block_decl()` über. Da alle Regeln der Syntax als virtuelle Funktionen ausgelegt sind, führt dies dazu, dass alle Aufrufe dieser Regel innerhalb von `CCSyntax` (und weiterer Basisklassen) zur neuen Implementierung verzweigen. Diese akzeptiert nun zusätzlich *Pointcut*- und *Advice*-Deklarationen. Die Regel `pointcut_decl()` ist erst auf der AspectC++-Ebene hinzugekommen. Sie implementiert die Regel *pointcut-declaration* aus der Grammatik. Dabei wird die Regel `decl()` aus der C++ Grammatik wiederverwendet.

Abbildung 7.6 auf der nächsten Seite zeigt die Struktur der Implementierung und den darin vorhandenen Datenfluss. Die Hauptarbeit wird durch PUMA übernommen, da die AspectC++ Erweiterungen an *Scanner*, *Parser* und semantischer Analyse vergleichsweise gering sind. Zudem sorgt PUMA für die Quellcodemanipulationen, die am Ende notwendig werden. Nicht durch die PUMA Bibliothek übernommen werden die Planung der Manipulationen (Planer) und die Durchführung der Transformationen (Weber). Bei der Planung wird beispielsweise ausgewertet, welche Verbindungspunkte zu einem *Pointcut* gehören, und entschieden, wie zu verfahren ist, falls mehrere Aspekte an derselben Stelle im Quellcode wirken müssen. Die Manipulationen sind komplexe Operationen wie das Weben von *Advice*-Code für eine bestimmte Art von Verbindungspunkt und dessen Aktivierung. Dabei werden diese komplexen Operationen auf die elementaren Manipulationen und teils auch Analysefunktionen des PUMA Systems heruntergebrochen.

Die gesamte AspectC++ Implementierung ohne die benutzte PUMA Bibliothek

```
// erweiterte C++-Regel
CTree *ACSyntax::block_decl () {
  return (parse (&ACSyntax::simple_decl) ||
          parse (&ACSyntax::asm_def) ||
          parse (&ACSyntax::pointcut_decl) ||
          parse (&ACSyntax::advice_decl)) ?
         builder ().block_decl () : (CTree*)0;
}
// neue Regel
CTree *ACSyntax::pointcut_decl () {
  return (parse (TOK_POINTCUT) &&
          parse (&ACSyntax::decl)) ?
         builder ().pointcut_decl () : (CTree*)0;
}
```

Abbildung 7.5: Ein Ausschnitt aus der `ACSyntax` Klasse

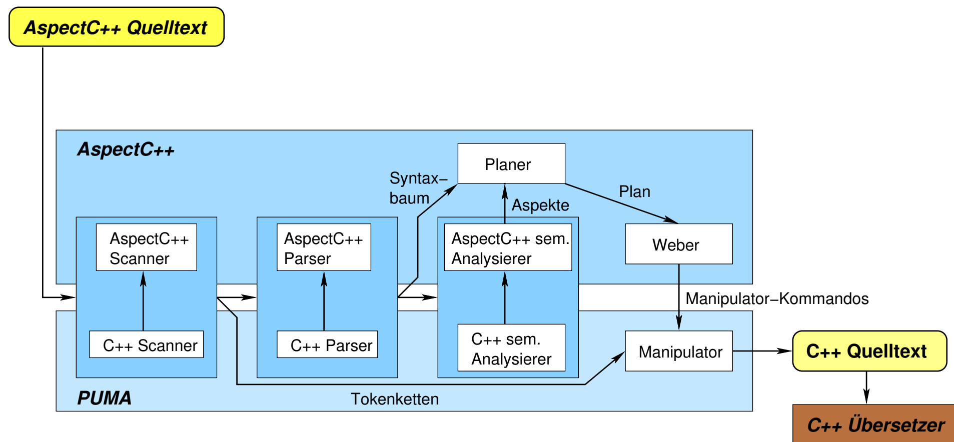


Abbildung 7.6: Struktur der AspectC++ Übersetzer Implementierung

umfasst derzeit etwa 7100 Zeilen Quelltext. Davon werden etwa 2800 zur Durchführung der Transformationen benötigt (`weaver`). Etwa 1500 Zeilen sind Analysecode, der der Planung zuzurechnen ist. 1600 Zeilen werden benötigt, um Datenstrukturen zu verwalten, die Aspekte, *Advice*, Einfügungen u.s.w. in Beziehung setzen. Circa 1000 Zeilen werden für die eigentliche Planung benötigt. Die restlichen Zeilen implementieren Hilfsklassen, zum Beispiel zur Analyse der Kommandozeilenparameter, die keiner der Hauptfunktionen direkt zugerechnet werden können.

7.4.2 Übersetzungseinheiten und Transformationsvorgang

In Abschnitt 7.2.1 wurde bereits angesprochen, dass für C++ Programmierer Übersetzungseinheiten und *Header*-Dateien eine wichtige Rolle spielen. Eine Erweiterung von C++ um Aspekte sollte in dieses existierende Schema hineinpassen. So wäre es unklug, die Codetransformation eines kompletten Projekts mit einem einzigen Aufruf von AspectC++ durchzuführen. Dies würde zwar den gedanklichen Aufwand beim Entwurf der Implementierung erleichtern, hätte jedoch den Nachteil, dass AspectC++ in existierende Werkzeugketten und integrierte Entwicklungsumgebungen nur schwer zu integrieren wäre. Ideal wäre es, wenn AspectC++ sich wie ein gewöhnlicher C++ Übersetzer verhalten würde, der pro Aufruf aus einer Übersetzungseinheit eine Objektdatei generiert. Leider reicht diese isolierte Sicht der Übersetzungseinheiten für AspectC++ nicht aus. Schließlich soll es auch möglich sein, Aspekte auf Übersetzungseinheiten wirken zu lassen, bei deren Erstellung noch nicht bekannt war, welche Aspekte einmal darauf wirken werden. Manuelle Erweiterungen durch den Programmierer sind dabei zu vermeiden.

Ein Kompromiss zwischen dem gleichzeitigen Zugriff auf alle Dateien eines Projektes und dem getrennten Zugriff auf die Übersetzungseinheiten besteht in der getrennten Bearbeitung der Übersetzungseinheiten unter Beachtung anderer Übersetzungseinheiten, die Aspekte enthalten. Theoretisch bedeutet dieses Vorgehen einen quadratischen Analyseaufwand, da für alle N Übersetzungseinheiten $N-1$ andere auf das Vorkommen von Aspektdefinitionen untersucht werden müssen. Dieser Aufwand lässt sich jedoch leicht reduzieren, wenn der Programmierer angibt, in welchen Übersetzungseinheiten überhaupt Aspekte definiert werden und welcher Aspekt auf welche Übersetzungseinheiten wirken können soll. Die erste Angabe erfolgt bei AspectC++ über Kommandozeilenparameter und die zweite mit Hilfe einer "#pragma weave" Direktive[6] in den jeweiligen Übersetzungseinheiten, die Aspekte enthalten. Durch diese Maßnahmen können Übersetzungseinheiten, auf die keine Aspekte wirken können, sehr schnell erkannt werden, so dass deren "Transformation" nur mit minimalem Zeitaufwand zu Buche schlägt.

Gleichzeitig kann bei Komponentencode, der nicht mit Hilfe von Namensräumen strukturiert wurde, eine Einschränkung des Zielbereichs von Aspekten auf Basis von Übersetzungseinheiten vorgenommen werden. Dies ist allerdings nur als

[6]Direktiven dieser Art werden auch bei C und C++ Übersetzern verwendet, damit der Programmierer dem Übersetzer Hinweise geben kann, die die Codegenerierung beeinflussen.

Notlösung anzusehen, da auf diese Weise Dateinamen plötzlich eine Relevanz für die Semantik eines Programms haben. Dies ist unüblich und sollte vermieden werden. “#pragma” Direktiven zur Optimierung oder zur Beschleunigung des Übersetzungsvorgangs sind dagegen gängige Praxis.

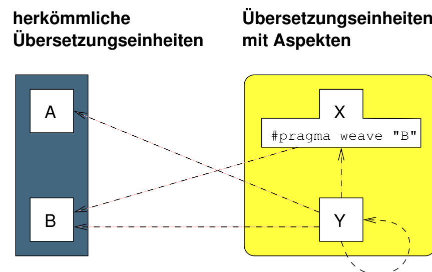


Abbildung 7.7: Potentielle Einwirkungsbeziehungen zwischen Übersetzungseinheiten

Abbildung 7.7 zeigt zur Verdeutlichung von “#pragma weave” vier Übersetzungseinheiten A, B, X und Y. Per Kommandozeilenparameter erklärt der Benutzer, dass X und Y Aspektdefinitionen enthalten. Soll nun beispielsweise A übersetzt werden, untersucht AspectC++ daraufhin X und Y auf “#pragma weave” Direktiven. Da innerhalb von X durch eine solche Direktive erklärt wird, dass die Aspekte dieser Übersetzungseinheit höchstens auf B wirken können, muss X bei der Übersetzung von A nicht berücksichtigt werden. Y dagegen enthält keine Einschränkung und muss nun durch den *Parser* analysiert werden, um zu ermitteln, ob irgendwelche Aspekte auf A wirken. Die entsprechenden Manipulationen in A werden daraufhin durchgeführt. Der Pfeil zwischen Y und A symbolisiert die soeben beschriebene Beziehung. Er zeigt zusammen mit den anderen Pfeilen in der Abbildung an, welche Übersetzungseinheiten mit ihren Aspekten potentiell auf welche anderen Übersetzungseinheiten wirken können. Wie zu erkennen ist, können Aspekte grundsätzlich in der eigenen Übersetzungseinheit wirken, sofern dies nicht durch ein “#pragma weave” ausgeschlossen wird. Neben dem einfachen Nennen eines Dateinamens können durch “#pragma weave” auch Gruppen von Übersetzungseinheiten über Muster selektiert werden.

7.4.3 Codegenerierung

Anhand des Beispielcodes in Abbildung 7.8 auf der nächsten Seite sollen einige spezielle Eigenschaften der Codegenerierung verdeutlicht werden. Es zeigt oben den Quelltext eines Aspekts, um Ausführungen der Methoden der Klasse `AClass` zu protokollieren. Unten ist der durch AspectC++ manipulierte Code dieser Klasse zu sehen. Wie zu erkennen ist, wurde die Funktion `func()` durch eine neue ersetzt, die vor der Ausführung der Originalfunktion, die in `__old_func()` umbenannt wurde, zunächst den *Advice*-Code aktiviert. Desweiteren wurde im Rahmen

Aspektcode:

```
#include <iostream>
aspect Trace {
   advice execution("% AClass::%(...)") : before () {
      cout << JoinPoint::signature() << endl;
   }
};
```

Die Klasse `AClass` *im generierten Code*:

```
class AClass {
   struct TJP_AClassfunc_FivE {
      inline static const char *signature () {
         return "int AClass::func()";
      }
      typedef ::AClass That;
      // ...
   };
   static inline void __advice_0_Trace_exec_AClassfunc_FivE(
      TJP_AClassfunc_FivE *thisJoinPoint) {
      cout << TJP_AClassfunc_FivE::signature() << endl;
   }
   int __old_func() { /* Originalcode der Funktion */ }
   inline int func () {
      int result;
      TJP_AClassfunc_FivE tjp_AClassfunc_FivE;
      __advice_0_Trace_exec_AClassfunc_FivE(&tjp_AClassfunc_FivE);
      result = AClass::__old_func ();
      return (int)result;
   }
};
```

Abbildung 7.8: Beispiel für die Codegenerierung des AspectC++ Übersetzers

der Codegenerierung eine Struktur namens `TJP_AClassfunc_FivE` angelegt. Sie dient der Beschreibung des hier manipulierten Verbindungspunktes. Die Funktion `__advice_0_Trace_exec_AClassfunc_FivE` ist eine manipulierte Kopie des *Advice*-Codes für diesen Verbindungspunkt. Der Name `JoinPoint` wurde in den Namen der generierten Beschreibungsklasse umbenannt und ein Parameter dieses Typs wird an sie übergeben.

Wichtig ist hier, dass die speziell für diesen Verbindungspunkt generierte Beschreibungsstruktur lediglich das eine Element enthält, das vom *Advice*-Code abgefragt wird. Es findet also eine Codegenerierung auf Anforderung statt, wodurch ein minimaler Ressourcenverbrauch erreicht wird.

Frühere Versionen von AspectC++ haben *Advice*-Code nicht explizit kopiert, sondern zu “inline” Funktionen des Aspekts gemacht, der durch Manipulation zu einer gewöhnlichen Klasse wurde. Die heutige Implementierung hat keine Nachteile bezüglich des Ressourcenverbrauchs, hat aber den Vorteil, dass zyklische Aufrufbeziehungen, wie sie in Abschnitt 7.2.3 besprochen wurden, vermieden werden, zu denen es früher leicht kam. Dazu musste lediglich der Aspektcode eine Methode der Klasse aufrufen, von der aus er aktiviert wurde. Das Ergebnis war nicht übersetzbarer C++ Code.

```
accode.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 0c                sub    $0xc,%esp
   7:   68 00 00 00 00          push   $0x0
                        8: R_386_32     .rodata
   c:   68 00 00 00 00          push   $0x0
                        d: R_386_32     cout
  11:   e8 fc ff ff ff          call   12 <main+0x12>
                        12: R_386_PC32  __ls__7ostreamPCc
  16:   89 04 24                mov    %eax,(%esp,1)
  19:   e8 fc ff ff ff          call   1a <main+0x1a>
                        1a: R_386_PC32  endl__FR7ostream
  1e:   83 c4 10                add    $0x10,%esp
  21:   89 d8                   mov    %ebx,%eax
  23:   8b 5d fc                mov    0xfffffffc(%ebp),%ebx
  26:   c9                      leave
  27:   c3                      ret
```

Abbildung 7.9: Generierter Maschinencode zum Beispiel in Abb. 7.8

Abschließend zeigt Abbildung 7.9 auf der vorherigen Seite den Maschinencode[7], der durch den C++ Übersetzer[8] in diesem Beispiel erzeugt wird, wenn die Funktion `main()` eine Instanz von `AClass` erzeugt und darauf die Methode `func()` aufruft. Durch den Mechanismus des Einbettens von Funktionen (*inlining*) werden sämtliche Funktionen aus dem Beispielcode direkt innerhalb der unvermeidbaren Funktion `main` ausgeführt. Die `push` Anweisung an Adresse 7 implementiert die Übergabe der konstanten Verbindungspunktsignatur aus dem *read-only* Datensegment an den Ausgabeoperator <<, der in diesem Fall aus einer Bibliothek stammt. Exakt der gleiche Maschinencode wird generiert, wenn der Beispielcode ohne den Aspekt `Trace` übersetzt, aber um die Ausgabe

```
cout << "int AClass::func()" << endl;
```

innerhalb von `main()` ergänzt wird. Diese Betrachtung zeigt, dass AspectC++ in Kombination mit einem optimierenden C++ Übersetzer die Bindung von *Advice*-Code ohne Verschwendung von Ressourcen implementiert.

7.5 Benutzung

Als ausführlicheres Beispiel wird an dieser Stelle die Implementierung des GNU C/C++ Sprachdialektaspekts für PUMA mittels AspectC++ erläutert. Dadurch werden die Ausführungen über PUMA in Kapitel 6 vervollständigt und gleichzeitig zeigt das Beispiel, welchen Nutzen AOP und AspectC++ in der praktischen Programmierung bringen kann.

Die Abweichungen des Sprachdialekts des GNU C/C++ Übersetzers betreffen sowohl die semantische Analyse, da beispielsweise Sichtbarkeitsbereiche von Variablen (*scopes*) bei bestimmten Kontrollstrukturen vom Standard abweichen, als auch die syntaktische Analyse, da auch hier erweitert wurde. Abbildung 7.10 auf der nächsten Seite zeigt zum Beispiel einen Ausschnitt aus der erweiterten Grammatik für `asm` Anweisungen beim GNU Dialekt im Vergleich zum Standard. Solche Anweisungen dienen der Einbettung von Assemblercodefragmenten in den C oder C++ Quelltext. Wie zu erkennen ist, muss für den Dialekt die Regel für *asm-definition* verändert werden und einige neue Regeln sind hinzuzufügen. Desweiteren ist dafür zu sorgen, dass der Builder, der bei PUMA für das Aufbauen des Syntaxbaums zuständig ist, um entsprechende Ausbaufunktionen erweitert wird und dass die notwendigen Klassen für die Syntaxbaumknoten vorhanden sind.

Abbildung 7.11 auf Seite 117 zeigt den Teil des GNU Dialektaspekts, der für diese Änderungen und Erweiterungen benutzt wird. Der Code der letzten *Advice*-Deklaration ermittelt, ob die GNU Erweiterungen des `asm()` Ausdrucks für den aktuellen *Parser*-Lauf zugelassen sein sollen. Dazu wird *Advice*-Code für die Konfigurierungsfunktion der Syntax Klasse definiert und über Kontextvariablen eine

[7] Intel x86 Assemblercode mit GNU Mnemonik, erzeugt mit `objdump -d --reloc`

[8] g++ 2.96 mit -O6 Codeoptimierung

Standard `asm` Syntax:

asm-definition:
  `asm ( `*string-literal*` ) ;`

GNU Dialekt `asm` Syntax:

asm-definition:
  `asm` *cv-qualifier*$_{opt}$ `(` *string-literal asm-extensions*$_{opt}$ `) ;`
asm-extensions:
  asm-operands
  asm-operands asm-operands
  asm-operands asm-operands asm-clobbers
asm-operands:
  : *asm-operand-list*$_{opt}$
asm-operand-list:
  ...

Abbildung 7.10: Syntax der `asm` Anweisung im GNU C/C++ Dialekt

Zugriffsmöglichkeit auf das Datenobjekt mit den Konfigurierungsparametern erlangt. Entsprechend dieser Parameter wird das Aspektattribut `extended_asm` gesetzt. Die Abfrage des Attributs erfolgt im Rahmen des Codes der vorletzten *Advice*-Deklaration. Sie sorgt dafür, dass die Ausführung der Funktion `asm_def()`, die die Regel *asm-definition* implementiert, in der Klasse `CSyntax` und allen abgeleiteten Klassen zum Aspekt umgeleitet wird. Der Code selbst überprüft das Attribut `extended_asm` und führt je nach dessen Belegung die Originalfunktion `asm_def()` oder die per Einfügung hinzugekommene Funktion `gnu_asm_def()` aus.

7.6 Zusammenfassung

Mit AspectC++ wurde eine aspektorientierte Spracherweiterung für C++ konzipiert und implementiert. Beim Entwurf der Sprache wurde versucht, die wesentlichen Sprachelemente von AspectJ in den C++ Kontext zu übertragen. Dabei mussten zum Beispiel Probleme wie das Alias-Problem und die Vermeidung zyklischer Klassenbeziehungen beachtet werden. Bei der Implementierung wurde höchster Wert darauf gelegt, dass möglichst viele Spracheigenschaften durch Codegenerierung und -manipulation und nicht durch ein Laufzeitsystem erbracht werden. Die Codegenerierung erfolgt auf Anforderung, so dass Anwender nur für die tatsächlich benutzen Spracheigenschaften einen Ressourcenverbrauch hinnehmen müssen. Damit sind die wesentlichen Voraussetzungen für den Einsatz aspektorientierter Implementierungstechniken und damit auch des aspektorientierten Entwurfs von Betriebssystemfamilien gegeben. Entsprechende Beispiele für den Einsatz von AspectC++ im Betriebssystembereich werden in Kapitel 9 vorgestellt.

AspectC++ erfordert keinerlei Preparierung des Komponentencodes, damit

```
#include "Puma/GnuCTree.h"  // GNU Syntaxbaumknotentypen
#include "Puma/Config.h"    // Zugriffsfunktionen auf
                              // Kommandozeilenparameter
aspect ExtGnuC {
  // Erweiterte asm Anweisungen zulassen?
  bool extended_asm;

  // Pointcut definitionen;
  pointcut syntax ()  = "CSyntax";
  pointcut builder () = "CBuilder";

  // Erweiterung der Syntax (Einfuegungen)
  advice syntax () : CTree * gnu_asm_def () {
    return (parse (TOK_ASM) && opt (parse (&CSyntax::cv_qual)) &&
            parse (TOK_OPEN_ROUND) && parse (&CSyntax::cmpd_str) &&
            opt (parse (&CSyntax::gnu_asm_operands) &&
                 parse (&CSyntax::gnu_asm_operands) &&
                 parse (&CSyntax::gnu_asm_clobbers)) &&
            parse (TOK_CLOSE_ROUND) && parse (TOK_SEMI_COLON)) ?
            builder ().gnu_asm_def () : (CTree*)0;
  }
  advice syntax () : CTree * gnu_asm_operands ()  { /* ... */ }
  // ...

  // Erweiterung des Builders (Einfuegungen)
  advice builder () : CTree * gnu_asm_def ()      { /* ... */ }
  advice builder () : CTree * gnu_asm_operands () { /* ... */ }
  // ...

  // Modifikation der Standard-Regel für asm Definitionen
  advice that (syntax) && execution ("% %::asm_def()") :
  around (CSyntax *syntax) {
    if (extended_asm)
      thisJoinPoint->result () = syntax->gnu_asm_def ();
    else
      thisJoinPoint->proceed ();
  }

  // Ermittlung der Konfigurationsparameter im Zuge der
  // Syntax-Konfigurierung
  advice args (config) && that (syntax ()) &&
         execution ("void %::configure(...)"):
  before (Config &config) {
    extended_asm = (config.Option ("--gnu") ||
                    config.Option ("--gnu-extended-asm"));
  }
};
```

Abbildung 7.11: Der GNU C/C++ Aspekt von PUMA

Aspekte darauf wirken können. Somit können die exisitierenden PURE Subsysteme auch für die Fallstudien mit AspectC++ verwendet werden und die Entwickler werden nur zum Umdenken gezwungen, wenn sie selbst Aspekte implementieren oder deren Implementierung durch Bereitstellung geeigneter Verbindungspunkte unterstützen wollen. Dies ist ein wesentlicher Vorteil gegenüber der Umsetzung von Aspekten mittels *Template*-Metaprogrammen oder eines Aspektmoderators. Abgesehen von den in Abschnitt 3.4 diskutierten Nachteilen dieser Ansätze wird dabei der Einsatz von Aspekten selbst zu einem *Crosscutting Concern*.

Kapitel 8

Weitere Aspektweber

Eine Vielzwecksspracherweiterung wie AspectC++ kann nicht alle denkbaren Arten von *Crosscutting Concerns* abdecken. So wurden in Abschnitt 5.2.1 drei Arten identifiziert, von denen AspectC++ nur für eine geeignet ist. Sollen die anderen Arten von *Crosscutting Concerns* aspektorientiert implementiert werden, bedarf es spezieller Aspektweber.

Dieses Kapitel stellt zwei solche Spezialweber vor, die beide auf PUMA basieren. Der erste Aspektweber ist COMA. Mit seiner Hilfe kann die Klassenstruktur einer Systemkomponente von einem separaten Aspektprogramm bestimmt werden, so dass die Lösung für das Problem, die "richtige" Implementierungsstruktur zu finden, bis zum Zeitpunkt der Systemkonfigurierung verschoben werden kann. Der zweite vorgestellte Aspektweber ist SOSP. Dieser Weber erlaubt es, die Strategie für Funktionseinbettungsentscheidungen (engl. *inline decisions*) vom eigentlichen Programmcode zu trennen.

8.1 COMA

Der Name COMA steht für PURE Component Manipulator. Dieses Werkzeug und das damit verbundene Konzept der "offenen Komponente", haben zum Ziel, das in Abschnitt 4.2.3 diskutierte Problem zu lösen, eine möglichst ressourcensparende Implementierungsstruktur für alle denkbaren Anwendungsszenarios bereitzustellen.

8.1.1 Generischer, flexibler oder spezifischer Programmcode

Um das Problem nochmals zu verdeutlichen, zeigt Abbildung 8.1 auf der nächsten Seite alternative Klassenstrukturen eines minimalen hypothetischen Speicherverwaltungssubsystems. Alle drei Strukturen in den Teilabbildungen (b) bis (d) passen zur funktionalen Hierarchie in Teilabbildung (a).

Die Strukturen in (b) und (c) sind flexibel und für verschiedene Anwendungsszenarios wiederverwendbar. Bei (b) erfolgt der Zugriff vom `HeapManager` auf die

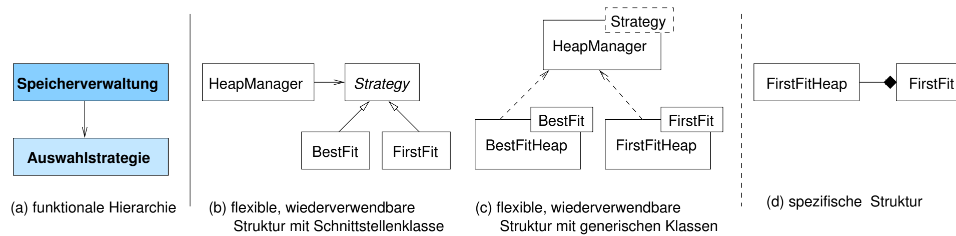


Abbildung 8.1: Eine funktionale Hierarchie und passende alternative Klassenstrukturen

konkreten Speicherverwaltungsstrategien indirekt über eine abstrakte Schnittstelle. Daraus resultiert dynamisches Binden, und das bedeutet wiederum, dass die Anwendung die Verbindung selbst zur Laufzeit entsprechend ihrer Anforderungen herstellen kann. Die Struktur in Teilabbildung (c) zeigt den `HeapManager` als generische Klasse (zum Beispiel ein C++ *Template*), die mit einer Strategie parametrisiert werden kann. Der Bindungszeitpunkt ist in diesem Fall der Übersetzungszeitpunkt der Anwendung. Während bei der ersten Variante die Flexibilität aus dem übersetzten Programmcode resultiert, werden bei der zweiten Variante einfach mehrere `HeapManager` Klassen generiert. Beide Strukturen sind flexibel genug, um sowohl einen `HeapManager` in Kombination mit der `BestFit` Strategie als auch mit der `FirstFit` Strategie bereitzustellen.

Teilabbildung (d) zeigt eine Struktur, bei der ein `HeapManager` fest mit der `FirstFit` Strategie verbunden ist. Hier erfolgt eine statische Bindung zum Zeitpunkt der Übersetzung der Klasse `HeapManager`. Diese Struktur stellt eine spezifische Lösung dar, die nur gewählt werden kann, falls feststeht, dass keine Anwendung einen `HeapManager` in Verbindung mit `BestFit` benötigen wird.

Tabelle 8.1 auf der nächsten Seite zeigt die Ergebnisse einer Codegrößenbestimmung im Rahmen eines einfachen Experiments[1]. Dabei wurden die Strukturen aus Abbildung 8.1 in entsprechende C++ Klassen umgesetzt. Die `BestFit` Strategie wurde nicht benutzt. Das heißt, in Variante 1 greift die Klasse `HeapManager` über eine abstrakte Basisklasse auf `FirstFit`Objekte zu, in Variante 2 wurde `HeapManager` zu einer *Template*-Klasse und in Variante 3 aggregiert ein `HeapManager` direkt ein `FirstFit` Objekt. Alle Klassen hatten bei dem Experiment jeweils drei Methoden zu etwa 100 Bytes. Zudem wurde aus den `HeapManager` Methoden heraus jeweils zweimal eine Methode der zugeordneten Strategie aufgerufen. Jede der Methoden des `HeapManager` wurde aus der Hauptfunktion `main()` heraus einmal referenziert.

Wie zu erkennen ist, lieferte bei dieser Messung die Variante 1 das schlechteste Ergebnis. Der Grund ist der Speicherplatzverbrauch für die virtuelle Funktionstabelle (Spalte “data”) und den zusätzlichen Code für den virtuellen Funktionsaufruf. Je kleiner die Funktionen in dem Beispiel sind, desto stärker macht sich der

[1] Verwendet wurde der g++ 2.96 unter Linux/x86. Die Größen beziehen sich auf die generierte Objektdatei, damit der *Startup*-Code und Bibliotheken auf die Größe keinen Einfluss haben.

Variante	Struktur	Name	code	data	bss	total
1	(b) links	abstrakte Basis	862	48	0	910
2	(b) rechts	*Template*	762	0	0	762
3	(c)	statische Bindung	762	0	0	762

Tabelle 8.1: Codegröße alternativer Klassenstrukturen bei einmaliger *Template* Instanziierung

zusätzliche Aufwand für das dynamische Binden bemerkbar. Dies geht bis zu einem Verhältnis von 84 Bytes zu 272 Bytes, wenn man die Größe der Funktionen schrittweise reduziert. Außerdem zeigt sich, dass die Variante 2 mit dem *Template* die gleiche Codegröße wie die Spezialstruktur von Variante 3 aufweist.

Bei einer zweiten Messung, deren Ergebnisse in Tabelle 8.2 dargestellt sind, wurde nun neben `FirstFit` auch die `BestFit` Strategie benutzt. Bei der *Template* Variante kommt es hier nun zur Codeduplikation. Daher ist in diesem Szenario die Variante 1 ein gutes Stück besser, was die Codegröße angeht. Der Unterschied kann sich noch drastisch verstärken, wenn die Funktionen zum Beispiel durch Einbettung von aufgerufenen Funktionen größer werden. Bei einer vierfachen Größe der Methoden der `HeapManager` Klasse liegt das Verhältnis beispielsweise schon bei 2029 Bytes zu 2969 Bytes.

Variante	Struktur	Name	code	data	bss	total
1	(b) links	abstrakte Basis	1197	72	0	1269
2	(b) rechts	*Template*	1433	0	0	1433
3	(c)	statische Bindung	—	—	—	—

Tabelle 8.2: Codegröße alternativer Klassenstrukturen bei zweifacher *Template* Instanziierung

Die hier gezeigten Messungen sollen deutlich machen, wie schwer es für einen Entwickler von wiederverwendbarer Software ist, die richtige Klassenstruktur zu finden, so dass der Ressourcenverbrauch für alle denkbaren Anwendungsszenarios möglichst minimal ist. In ungünstigen Fällen kann die Ressourcenverschwendung eine signifikante Größe annehmen.

Die Spezialstruktur ist auf jeden Fall ressourcensparend, aber nicht wiederverwendbar. Die Entscheidung zwischen generischem und flexiblem Code erfordert ebenfalls Wissen über das Anwendungsprofil. So müsste der Entwickler wissen, wie oft der generische Code instanziiert werden müsste.

8.1.2 Offene Komponenten

Das Konzept der offenen Komponenten [44, 40, 45] hat zum Ziel, das im letzten Abschnitt beschriebene Problem zu vermeiden, indem die Entscheidung über die letztendliche Struktur verschoben wird, bis das System unter Zuhilfenahme von Anwendungswissen konfiguriert wird. Damit wird der implementierte Quellcode zu einem Bauplan, aus dem sich viele verschiedene Systeme mit unterschiedlichen Klassenstrukturen herleiten lassen. Der Name "offene Komponente" begründet sich auf die Anpassbarkeit zum Zeitpunkt der Integration in ein Anwendungsprojekt (engl. *deployment time*), die bei den klassischen binären Komponenten [103] nicht gegeben ist.

Eine offene Komponente wird dabei zweigeteilt aufgefasst. Neben dem Quellcode, der die eigentliche Funktionalität erbringt, gibt es Aspektcode, der die implementierte Struktur beschreibt. Dazu kommt weiterer Aspektcode für die Formulierung der Strukturanforderungen im jeweiligen Anwendungsszenario. Der Aspektweber COMA sorgt dafür, dass vor dem Übersetzen eine Systemstruktur hergestellt wird, die alle Anforderungen erfüllt, gleichzeitig aber so sparsam wie möglich mit den Ressourcen umgeht. Durch die automatisierte Transformation wird ein Maß an Robustheit bei der Integration erreicht, was eine der wichtigsten Forderungen an Softwarekomponenten ist [104].

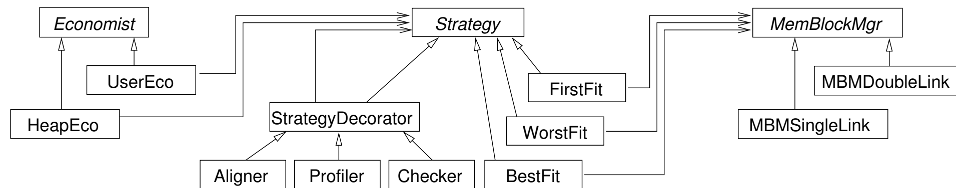


Abbildung 8.2: Klassenstruktur einer Speicherverwaltungskomponente

Als Beispiel wird nun eine etwas komplexere Speicherverwaltungskomponente betrachtet. Um eine höchstmögliche Wiederverwendbarkeit zu erreichen, folgt der Entwurf der Forderung, dass als Bindungszeitpunkt von Objekten der späteste in der Anwendungsdomäne denkbare Zeitpunkt zu wählen ist. Das Ergebnis ist in Abbildung 8.2 zu sehen. Diese Entwurfsregel führt dazu, dass fast grundsätzlich auf dynamisches Binden gesetzt wird, wodurch gemeinsame Schnittstellen von Klassen in Form abstrakter Basisklassen im Klassendiagramm zu erfassen sind.

Würde nun ein solcher Entwurf direkt in eine Implementierung umgesetzt werden, käme es zu einem Maximum an Flexibilität. Klassische Entwurfsmuster [46] können eingesetzt werden, um solche Strukturen zu finden, da auch sie sehr stark auf explizit sichtbare Schnittstellen und dynamisches Binden setzen. Bei der Speicherverwaltungskomponente wurde zum Beispiel das *Strategy*-Muster und das *Decorator*-Muster angewendet.

Wie der letzte Abschnitt gezeigt hat, führt flexibler Code oft zu Ressourcenverschwendung. Daher sorgt der konfigurierbare Teil des Aspektcodes dafür, dass

der Bindungszeitpunkt für bestimmte Objektbeziehungen vorverlegt werden kann. Daneben wird festgelegt, welche Klassen der Komponente überhaupt benötigt werden. Abbildung 8.3 zeigt, wie auf diese Weise aus der entworfenen Struktur in Abbildung 8.2 mehrere anwendungsspezifische Subsystemstrukturen erzeugt werden können. Technisch gesehen manipuliert der Aspektweber den Quellcode, der der flexiblen Entwurfsstruktur entspricht, so dass entsprechend der jeweiligen Strukturanforderung zum Beispiel eine der Strukturen auf der rechten Seite der Abbildung entsteht.

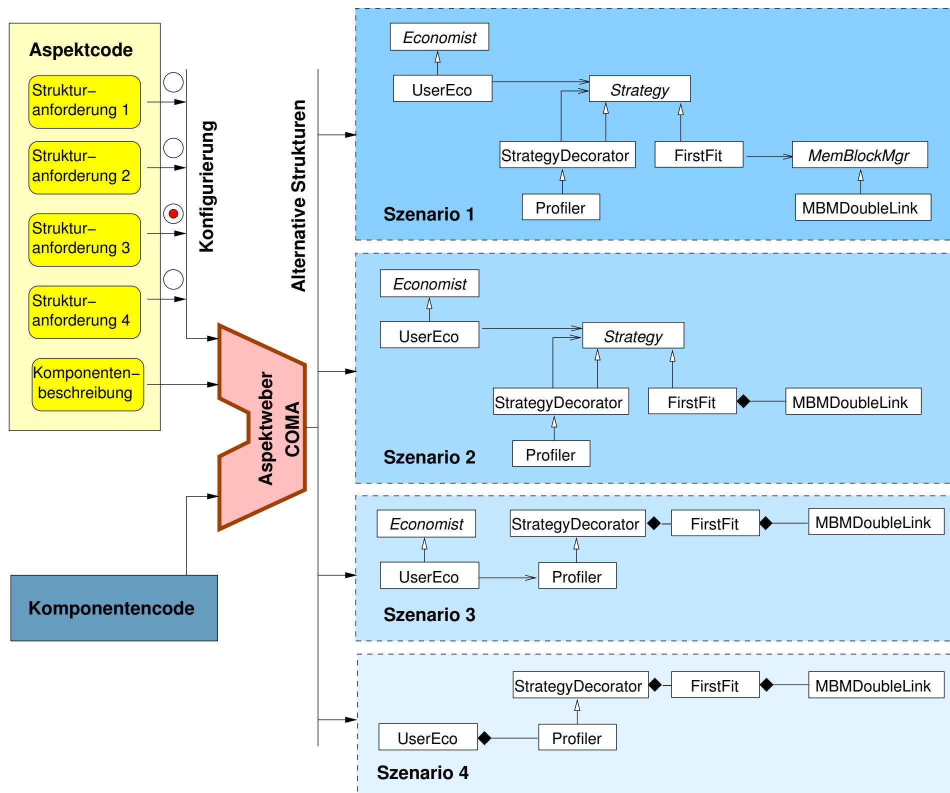


Abbildung 8.3: Restrukturierung einer Speicherverwaltungskomponente

Natürlich bringt es zahlreiche Probleme mit sich, unabhängig entwickelte Quellcodekomponenten im Rahmen eines Anwendungsprojekts zu integrieren oder auch nur für sich allein zu testen. So kann es bei der Integration zu Namenskonflikten kommen und Komponenten können Funktionen nutzen, die von anderen Komponenten erbracht werden müssen. Da all diese Gesichtspunkte jedoch weniger mit AOP als mit dem Komponentengedanken zu tun haben, sei an dieser Stelle nochmals auf die entsprechenden Publikationen verwiesen [44, 40, 45]. An dieser Stelle wird lediglich die sprachliche und technische Umsetzung sowie das Ergebnis solcher Restrukturierungen betrachtet.

8.1.3 Sprachliche und technische Umsetzung

Für die Beschreibung der aus dem Entwurf resultierenden flexiblen Komponentenstruktur des vom Entwickler gelieferten Implements und der Strukturanforderungen wird auf die Sprache XML zurückgegriffen. Sie eignet sich durch ihre Flexibilität für verschiedenste Arten von strukturierten Dokumenten und zahlreiche Werkzeuge unterstützen den Umgang damit.

```
<?xml version="1.0"?>
<!DOCTYPE component PUBLIC "-//PURE//DTD COMDEL 1.0//EN" "http: ..."
<component name="mem">
  <parts>
    <class name="Strategy">
    ...
   <class name="UserEco">
      <sources>
        <file name="UserEco.h" type="def"/>
        <file name="UserEco.cc" type="impl"/>
      </sources>
      <attr name="memory" type="void * %" visible="private"/>
      <attr name="bytes" type="unsigned long %" visible="private"/>
      <rel name="Strategy" type="ref" attr="strategy"/>
      <attr name="strategy" type="Strategy * %" visible="private"/>
      <rel name="Economist" type="base"/>
      <method name="setup" type="void %" visible="public">
        <param name="strat" type="Strategy * %"/>
        <param name="mem" type="void * %"/>
        <param name="size" type="unsigned long %"/>
      </method>
      <method name="strat" type="Strategy * %" visible="public"/>
      <method name="malloc" type="void * %" visible="public">
        <param name="size" type="unsigned long %"/>
      </method>
      <method name="free" type="void %" visible="public">
        <param name="mem" type="void * %"/>
      </method>
    </class>
  </parts>
</component>
```

Abbildung 8.4: Eine XML-basierte Komponentenbeschreibung

Abbildung 8.4 zeigt einen Auszug aus der XML Beschreibung der Klassenstruktur aus Abbildung 8.2 auf Seite 122. Darin werden alle Klassen der Komponente und deren Beziehungen aufgeführt. Solche Beschreibungen können entweder direkt von einem computergestützten Entwurfswerkzeug (*CASE Tool*) generiert werden oder sie werden, wie in diesem Beispiel geschehen, anhand einer Codeana-

lyse automatisch ermittelt. Für den Entwickler fällt in jedem Fall keine zusätzliche Arbeit an.

Welche Struktur eine Komponente dann tatsächlich einnehmen soll, muss bei deren Integration in eine konkrete Anwendung festgelegt werden. Ein familienbasierter Entwurf stellt gegebenenfalls je nach Familienmitglied verschiedene Anforderungen an die Komponente. Bie PURE werden diese Strukturanforderungen mit Hilfe der selektierten Systemeigenschaften aus dem Feature Modell gebildet.

```
<requirement component="MemoryManagement">
   <class name="UserEco" binding="aggregate">
      <relation name="strategy" type="aggregate" class="Profiler">
   </class>
   <class name="StrategyDecorator">
      <relation name="strategy" type="aggregate" class="FirstFit">
   </class>
   <class name="FirstFit" binding="aggregate">
      <relation name="blockmgr" type="aggregate"
                class="MBMDoubleLink">
   </class>
</requirement>
```

Abbildung 8.5: Eine Strukturanforderung für eine offene Komponente

Eine mögliche Strukturanforderung ist in Abbildung 8.5 zu sehen. Mit dieser Anforderung würde COMA Code generieren, der Variante 4 aus Abbildung 8.3 auf Seite 123 entspricht. Diese Struktur ist besonders ressourcensparend, da keine einzige virtuelle Funktion benötigt wird. Die erste Zeile sagt aus, dass hier eine Anforderung an die Struktur der Komponente MemoryManagement gestellt wird. Danach folgen drei Blöcke, die sich auf Klassen dieser Komponente beziehen. Einerseits drücken sie aus, dass die entsprechenden Klassen benutzt werden sollen, andererseits können mit <relation name=...> Beziehungen dieser Klasse verändert werden. So wurde hier zum Beispiel die Beziehung namens strategy der Klasse UserEco, die in der Komponentenbeschreibung eine Beziehung zur abstrakten Basisklasse Strategy ist, in eine Aggregationsbeziehung mit der davon abgeleiteten Klasse Profiler umgewandelt. Auf diese Weise wird der Bindungszeitpunkt zum Zeitpunkt der Komponentenübersetzung vorverlegt.

8.1.4 Ergebnis

Tabelle 8.3 auf der nächsten Seite zeigt die Ergebnisse des Ansatzes, Systemkomponenten anwendungsspezifisch zu strukturieren[2]. Für diese Messung wurde die

[2]Die Übersetzung erfolgt mit dem g++ 2.96. Bei der Laufzeitmessung wurde jeweils ein Block von 10 Bytes belegt und wieder freigegeben. Beim AVR erfolgte die Angabe in Taktzyklen. Bei der x86 Messung wurde dieser Vorgang 1000-mal auf einem AMD Athlon 700 wiederholt. Die Angabe

Variante	Intel x86					
	text	data	total	%	time	&
1	1304	224	1528	100	260	100
2	984	116	1100	72	191	73
3	705	48	753	49	146	56
4	508	0	508	33	78	30

Variante	Atmel AVR					
	text	data	total	%	time	%
1	2208	372	2580	100	862	100
2	1784	193	1977	76	663	77
3	1362	51	1427	55	484	56
4	966	0	966	37	318	37

Tabelle 8.3: Codegrößen und Laufzeiten verschiedener Speicherverwaltungssystemvarianten

Speicherverwaltungskomponente entsprechend dem Klassendiagramm in Abbildung 8.2 auf Seite 122 implementiert. Dazu wurde ein Anwendungsprogramm erstellt, das von seinen Anforderungen her mit der Struktur von Variante 4 aus Abbildung 8.3 auskäme. Nun wurden vier Strukturanforderungen entsprechend Variante 1 bis 4 erstellt und nach der Codemanipulation durch COMA wurde die resultierende Komponente jeweils zusammen mit der Anwendung zu einem fertigen Programm gebunden.

Wie aus der Tabelle abzulesen ist, konnte aufgrund der Anpassung im Vergleich zu der flexibelsten Variante der Ressourcenverbrauch an Speicher als auch an Rechenzeit auf etwa ein Drittel reduziert werden.

8.2 SOSP

Die Abkürzung SOSP steht für *Source Splitter*. Dieser Name ist historisch begründet, da die erste Aufgabe dieses Werkzeugs darin bestand, Quelltexte so zu zerlegen, dass jeder Programmcode einer Funktion eine eigene Übersetzungseinheit bildet. Auf diese Weise kann der sonst dazu nicht fähige GNU Binder `ld` dazu gebracht werden, auf Funktionsbasis zu arbeiten. Dadurch wird vermieden, dass unbenutzte Funktionen eingebunden werden, nur weil eine andere Funktion innerhalb derselben Übersetzungseinheit referenziert wird.

Die zweite sehr wichtige Aufgabe von SOSP besteht in der Implementierung einer Strategie für Funktionseinbettungsentscheidungen. Die folgenden Abschnitte erläutern diese Strategie, die Struktur und Benutzung von SOSP sowie die erreichten Ergebnisse.

erfolgt in μs.

8.2.1 Die Funktionseinbettungsstrategie

In C++ kann ein Programmierer Einfluss darauf nehmen, ob beim Aufruf einer Funktion der Übersetzer eine Sprunginstruktion erzeugen soll, oder ob der Code der aufgerufenen Funktion direkt an der Aufrufstelle eingebettet werden soll. Abbildung 8.6 zeigt, wie diese Einflussnahme unter Verwendung des Schlüsselwortes `inline` erfolgt.

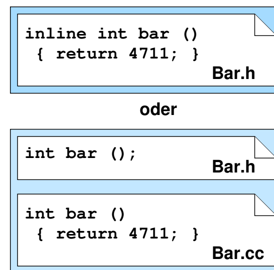


Abbildung 8.6: Benutzung des `inline` Schlüsselworts

Zwar ist der Übersetzer laut Sprachstandard nicht daran gebunden, dem Hinweis durch das `inline` Schlüsselwort auch zu folgen, doch typischerweise wird davon nur abgewichen, wenn der Quellcode in der Übersetzungseinheit nicht sichtbar ist oder es sich um eine virtuelle Funktion handelt. Damit hat der Programmierer hier eine Entscheidung zu treffen, die großen Einfluss auf den Ressourcenverbrauch eines Systems haben kann. Schließlich kommt es bei mehrfachem Einbetten von Funktionen zu Codeduplikation. Da Einbettungsentscheidungen für alle Funktionen zu treffen sind, ist ein Programmierer gezwungen, eine Strategie für die Entscheidungen zu wählen. Da die Umsetzung dieser Strategie jede Funktion betrifft, handelt es sich dabei um ein *Crosscutting Concern*.

Die Qualität einer Einbettungsentscheidung für eine Funktion hängt von folgenden Faktoren ab:

- Größe der Funktion: Bei großen Funktionen macht sich Codeduplikation stärker bemerkbar als bei kleinen.
- Gesamtanzahl der Aufrufe: Der Speicherplatzverbrauch einer Funktion multipliziert sich bei der Einbettung mit der Anzahl der Funktionsaufrufe
- Andere Einbettungsentscheidungen: Wenn eine Funktion andere aufruft und diese wiederum eingebettet werden, wird die aufrufende Funktion größer.
- CPU Typ bzw. Übersetzer: Gewinne können durch das Einbetten erzielt werden, da kein Funktionsprolog und -epilog generiert werden müssen und kompakterer Code entsteht, weil die Verwendung von Registern im Allgemeinen bei eingebetteten Funktionen besser optimiert werden kann. Wie groß diese

Vorteile des Einbettens im Vergleich zu dem Nachteil der Codeduplikation sind, hängt sehr stark vom Übersetzer ab.

All diese Punkte können von einem Entwickler einer einzelnen Betriebssystemkomponente, die in einer Familie Verwendung finden soll, nicht beurteilt werden. So ist die Anzahl der Aufrufe aus Sicht der einzelnen Komponente unbekannt, da diese stark von der Konfiguration abhängt. Auch der Übersetzer ist nicht bekannt, da verschiedenste Zielplattformen unterstützt werden sollen.

Dieses Problem kann gelöst werden, indem die Einbettungsstrategie auch einer Konfigurierung unterzogen wird. Damit werden die notwendigen Entscheidungen herausgezögert, bis das notwendige Wissen vorliegt. Für eine Konfigurierung bietet sich eine aspektorientierte Implementierung der Strategie an, da so nur das Aspektprogramm auszutauschen ist und nicht ein Eingriff pro Funktion erfolgen muss. SOSP erlaubt solch eine aspektorientierte Strategieimplementierung.

8.2.2 Struktur und Benutzung

Abbildung 8.7 zeigt, wie SOSP mit konfigurierbaren Strategie-Skripten zusammenarbeitet. SOSP hat in dem Zusammenspiel die Aufgabe, Informationen über das konfigurierte System zu sammeln und diese als Entscheidungshilfen den Skripten zur Verfügung zu stellen. Die dort gelieferten Informationen umfassen die Liste der Funktionen, eine Abschätzung ihrer Größe, den Aufrufgraphen und den gewählten Übersetzer. Die Aufgabe eines Strategie-Skripts besteht nun darin, auf Basis dieser Informationen und eventuell weiterer externer Informationen Einbettungsentscheidungen zu erstellen. So wird festgelegt, welche Funktionen des Systems beim Aufruf einzubetten sind. Mit Hilfe einer Systemmanipulation sorgt SOSP dann dafür, dass die Übersetzung entsprechend der strategischen Entscheidungen erfolgt.

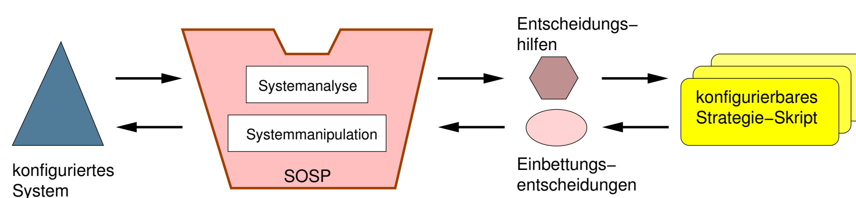


Abbildung 8.7: Globale Einbettungsstrategien durch SOSP und Strategie-Skripte

Die Kommunikation zwischen SOSP und den Strategie-Skripten erfolgt über Textdateien. Damit können die Skripte in jeder beliebigen Programmiersprache verfasst werden, die dem Strategieentwickler problemadäquat erscheint. Denkbar sind zum Beispiel folgende Strategien:

- Heuristiken auf Basis der SOSP Entscheidungshilfen

  - Funktionen bis zu einer bestimmten Größe werden eingebettet
  - Funktionen, die mehr als einmal aufgerufen werden, dürfen nicht eingebettet werden

- Heuristiken unter Beachtung der Anwendung oder eines Anwendungsprofils
- Suchstrategien zur Findung guter Einbettungsentscheidungen durch Übersetzen und Messen
  - "brute force" Suche
  - Genetische Suche, bei der jeweils erfolgreiche Einbettungsentscheidungen miteinander kombiniert werden, um noch bessere zu finden.
- Manuelles Festlegen der Entscheidungsmenge oder eines Teils davon

All diese Strategien können auch miteinander kombiniert werden. Nur durch die Anwendung von SOSP wurde es möglich, mit solchen Strategien zu experimentieren, da sonst die verwendete Strategie untrennbar mit jeder Funktion verbunden wäre.

8.2.3 Ergebnis

Um die Vorteile des mit SOSP verfolgten aspektorientierten Ansatzes zu zeigen, wurden vier PURE Systeme anwendungsspezifisch konfiguriert und dann mittels SOSP verschiedenen Funktionseinbettungsstrategien unterworfen. Im Folgenden werden die vier Anwendungsszenarien kurz beschrieben:

philo: In diesem Szenario synchronisieren sich fünf Programmfäden mittels Semaphoren entsprechend einer Lösung des bekannten Problems der speisenden Philosphen [105]. Die CPU Zuteilung erfolgt hier unterbrechungsgesteuert nach dem *Round-Robin* Prinzip.

candidature: Diese Anwendung erzeugt verschiedene Fäden, die in einer Schleife Ausgaben machen und dann jeweils freiwillig die CPU abgeben. Die CPU Zuteilung erfolgt kooperativ auf Basis von Prioritäten.

watchdog: Hier wird eine Unterbrechungsbehandlungsroutine definiert und an den Zeitgeber gebunden. Durch eine entsprechende Programmierung des Zeitgebers wird die Behandlungsroutine in regelmäßigen Abständen aktiviert. Die Fadenverwaltung wird hier nicht benötigt.

io: In diesem Szenario werden verschiedene Funktionen der PURE Ein-/Ausgabe Bibliothek aufgerufen. Es handelt sich um eine Testanwendung die die wichtigsten Fälle abdeckt.

Bei den Einbettungsstrategien wurden drei Gruppen von Heuristiken verwendet. Die erste Gruppe erfordert lediglich lokales Wissen über eine Funktion, um eine Entscheidung zu treffen. Die Gruppe umfasst die Strategien "None", bei der keine Funktion eingebettet wird, "All", bei der alle Funktionen eingebettet werden sollen und "L-*K*". Letztere sorgt dafür, dass alle Funktionen eingebettet werden sollen, deren Anzahl der Zeilen im Quelltext kleiner als K ist. Die Idee dabei ist, dass nur bei kleinen Funktionen der Gewinn durch die Vermeidung von Funktionsprologen und -epilogen und eine bessere Registerbelegung die Nachteile durch Codeduplikation ausgleicht.

Die zweite Gruppe umfasst die Strategien "LV-*K*". "LV-*K*" und "L-*K*" sind identisch bis auf die Tatsache, dass bei "LV-*K*" virtuelle Funktionen nie eingebettet werden, auch wenn diese mit einem voll qualifizierten Namen aufgerufen werden, so dass eine Einbettung prinzipiell möglich wäre. Auf den ersten Blick wird für diese Strategien auch nur lokales Wissen verwendet. Wenn jedoch für die Implementierung der Systemkomponenten ein Werkzeug wie COMA benutzt wird, kann erst zum Zeitpunkt der anwendungsspezifischen Konfigurierung festgestellt werden, ob eine Funktion virtuell ist.

Bei der dritten Gruppe handelt es sich um Heuristiken, die nach der Konfigurierung des Systems den Aufrufgraphen untersuchen, um die Kosten einer Einbettungsentscheidung möglichst gut abzuschätzen. Bei "Tree" wird die Einbettungsentscheidung anhand der folgenden Funktion $Tree(f)$ getroffen:

$$Tree(f) = \begin{cases} ja & \text{falls } lines(f) \leq 4, \\ nein & \text{sonst falls } virtual(f), \\ ja & \text{sonst falls } calls(f) = 1 \vee (calls(f) = 2 \wedge lines(f) \leq 6), \\ nein & \text{sonst} \end{cases}$$

Dabei ist f die gerade betrachtete Funktion und $lines(f)$ die Anzahl ihrer Quelltextzeilen. Das Prädikat $virtual(f)$ gibt an, ob f virtuell ist, und $calls(f)$ ist die Anzahl der Aufrufe der Funktion. Mit den Entscheidungen wird bei den Blättern des Aufrufgraphen begonnen. Nach jeder Entscheidung für eine Einbettung wird die Anzahl der Zeilen aller Funktionen, die die aktuell betrachtete Funktion aufrufen, um $lines(f)$ erhöht, so dass für diese Funktionen fortan eine höhere Zeilenzahl angenommen wird. Eine weitere Heuristik "TreeV" entspricht "Tree", nur dass virtuelle Funktionen grundsätzlich nicht eingebettet werden.

Abbildung 8.8 auf der nächsten Seite zeigt nun die Ergebnisse des Tests. Dort ist für die vier Anwendungsszenarien jeweils dargestellt, wie groß die Verbesserung der Codegröße[3] unter Anwendung der jeweiligen Heuristik ist. Unter der "Verbesserung" wird hier die Verkleinerung der Codegröße relativ zur durchschnittlich erzielten Codegröße verstanden.

Bei den oberen beiden Szenarien ergibt sich ein recht einheitliches Bild. In beiden Fällen führt das Einbetten aller Funktionen oder das Einbetten keiner Funk-

[3] Die Summe der Größen des Text-, Daten- und BSS Segments, wobei die Größe der für Anwendungsfäden benötigten Stapelspeicher abgezogen wurde. Die Messungen erfolgten auf Basis der PURE/x86 Variante mit dem gcc 2.96.

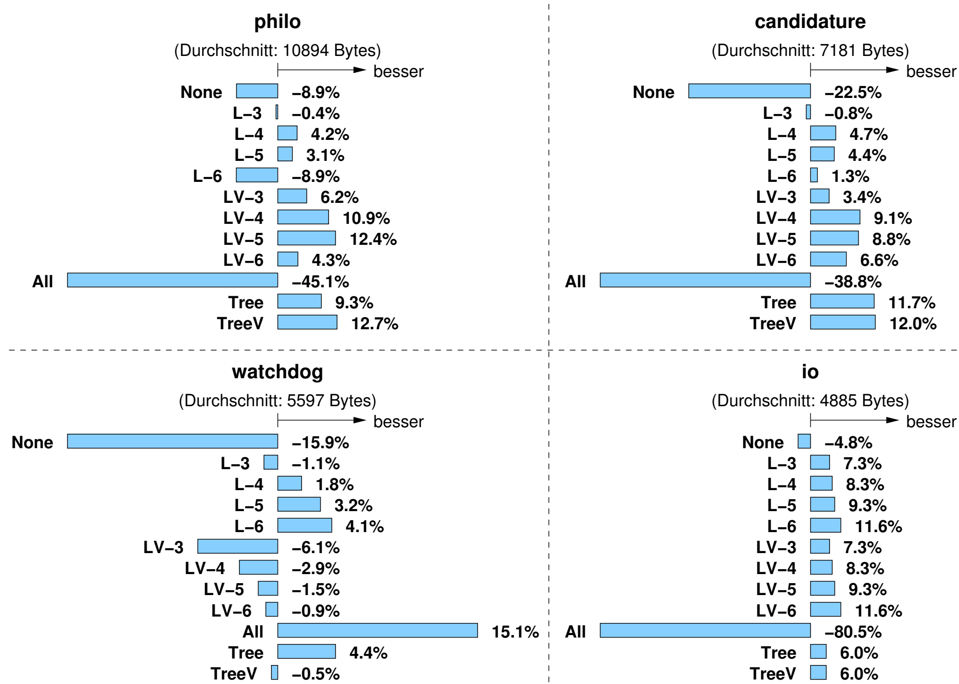


Abbildung 8.8: Verbesserung der Codegrößen verschiedener Systeme im Vergleich zum Durchschnitt bei Anwendung unterschiedlicher Einbettungsstrategien

tion zu schlechten Ergebnissen. Weiterhin zeigt sich, dass “LV-K” im Vergleich zu “L-K” und “TreeV” im Vergleich zu “Tree” besser abschneidet. Das heißt, dass virtuelle Funktionen besser nicht eingebettet werden sollten. Die besten Ergebnisse für “L-K” und “LV-K” sind für K=4 oder K=5 festzustellen. Die generelle Tendenz ist, dass das Ergebnis umso besser ist, je mehr Wissen für die Heuristik verwendet wurde. Das beste Ergebnis liefert in beiden Fällen “TreeV”. Damit wird die These unterstützt, dass Einbettungsentscheidungen nicht durch den Programmierer einer Systemkomponente, sondern im Rahmen des Konfigurierungsvorgangs mit Hilfe einer globalen Strategie erfolgen sollten. Erst zu diesem Zeitpunkt ist das vollständige Wissen über den Aufrufgraphen und die Größe der Funktionen vorhanden.

Die beiden unteren Szenarien zeigen ein anderes Bild. Es handelt sich hier um zwei Sonderfälle. Bei dem Szenario “watchdog” sind vergleichsweise viele virtuelle Funktionen im System vertreten, deren Aufrufe oft auch sinnvoll eingebettet werden könnten. Dadurch sind hier “LV-K” und “TreeV” relativ schlecht. Das Überraschende an diesem Beispiel ist, dass die aggressive Strategie des Einbettens aller Funktionen mit Abstand zum besten Ergebnis führt.

Im Szenario “io” sind keine virtuellen Funktionen im System vertreten. Da so der Übersetzer bei jedem Funktionsaufruf eine Einbettung vornehmen kann, ergibt sich bei “All” ein katastrophales Bild. “L-K” und “LV-K” sowie “Tree” und “TreeV” liefern jeweils das gleiche Ergebnis. Das Überraschende ist hier, dass die mit viel Wissen arbeitenden Heuristiken “Tree” und “TreeV” relativ schlecht abschneiden. Dies lässt sich nur damit erklären, dass das Beispiel insgesamt relativ klein ist, so dass bereits wenige zufälligerweise richtige Entscheidungen auf Seiten der unwissenden Heuristiken das Ergebnis stark positiv beeinflussen können.

Die beiden unteren Szenarien zeigen, dass keine der hier vorgestellten Einbettungsstrategien in jedem Fall die besten Ergebnisse liefert. Gerade bei kleinen Systemkonfiguration kann es zu Überraschungen kommen. Der Vorteil der aspektorientierten Herangehensweise an das Problem der Einbettungsentscheidungen zeigt sich hier deutlich. Es ist nun ohne großen Aufwand möglich, zu erkunden, mit welcher Strategie im jeweiligen Szenario die besten Ergebnisse erreicht werden. Diese kann dann für das fertige System gewählt werden. Darüber hinaus kann eine Untersuchung des resultierenden Systems für eine ergänzende Feinabstimmung herangezogen werden, indem die Entscheidungsmenge mit manuellen Vorgaben verknüpft wird. Neben der Suche nach redundantem Code können dazu auch Laufzeitmessungen herangezogen werden. Dies ist sinnvoll, wenn Speicherplatzverschwendung im Austausch gegen eine schnellere Programmausführung hingenommen wird. Wichtig ist, dass dank SOSP all diese Entscheidungen keine Auswirkungen auf den Quelltext der Systemkomponenten haben und somit leicht änderbar und durch die modulare Implementierung gut nachvollziehbar sind.

8.3 Zusammenfassung

Wie die Ergebnisse zeigen, sind COMA und SOSP sehr wichtige Werkeuge für die Implementierung von wiederverwendbaren Komponenten von Programmfamilien. Sie tragen dazu bei, dass der Ressourcenbedarf mit den Anwendungsanforderungen skaliert. Beide erreichen dies durch einen aspektorientierten Ansatz. Sie erlauben die separate und modulare Implementierung eines *Crosscutting Concerns*, wodurch eine leichte Konfigurierbarkeit erreicht wird.

Im Prinzip könnten Übersetzererweiterungen diese beiden Aspektweber unnötig machen. Dazu müssten allerdings folgende Eigenschaften gegeben sein:

- Es müsste globales Wissen genutzt werden, statt nur einzelne Übersetzungseinheiten zu analysieren. Einige moderne Übersetzer wie der GNU und Intel C++ Übersetzer gehen derzeit in diese Richtung.
- Die Trennung zwischen der Übersetzung einer Bibliothek und der Anwendung der Bibliothek müssten beachtet werden.
- Es müsste möglich sein, strategische Entscheidungen wie die der Funktionseinbettungen von außen projektspezifisch zu beeinflussen.

Wären diese Eigenschaften gegeben, dann könnte der Übersetzer die Rolle des Aspektwebers übernehmen.

Beide hier vorgestellten Aspektweber sind interessant für die Entwicklung von Programmfamilien, die in C++ implementiert werden. Sie haben jedoch keine spezielle Relevanz in Bezug auf Betriebssysteme. Daher wird im Rahmen der Fallstudien nicht weiter darauf eingegangen. Wichtig ist jedoch zu wissen, dass bei allen Messergebnissen, die im nächsten Kapitel vorgestellt werden, das Einbettungsverhalten von SOSP und einem Skript für die Strategie “L-5” bestimmt wurde.

Kapitel 9

Fallstudien

In diesem Kapitel werden drei Fallstudien präsentiert, die zeigen sollen, ob die Verbindung von Aspektorientierung und Programmfamilien im Betriebssystembau tatsächlich die erhofften Vorteile bringt. Die Bewertungskriterien wurden am Ende von Kapitel 5 auf Seite 65 aufgelistet.

Bei der ersten Fallstudie geht es vor allem um den Entwurf. Basierend auf einem Feature-Modell und einer *Concern*-Hierarchie wird eine Klassen/Aspekt-Struktur ermittelt, mit der abstrakte Prozesszustände in einer Prozessverwaltung modular implementiert werden können. Es handelt sich also um ein *Crosscutting Concern* mit relativ begrenzter Wirkung. In diesem Beispiel werden einige immer wiederkehrende Vorteile und auch Probleme angesprochen, vor denen Entwickler beim Entwurf einer Programmfamilie stehen, und inwieweit sie nun lösbar sind.

Die zweite Fallstudie ist noch stärker betriebssystembezogen. Sie stellt die aspektorientierte Implementierung der Unterbrechungssynchronisation in PURE dar. Es handelt sich dabei um ein *Crosscutting Concern*, das mehrere Subsysteme übergreift. Damit wird gezeigt, dass es tatsächlich möglich ist, globale Strategien in Betriebssystemen modular zu implementieren und damit leicht austauschbar zu machen.

Mit der dritten Fallstudie soll gezeigt werden, welchen Einfluss Aspekte auf die Architektur eines Betriebssystems haben können. Es handelt sich dabei um die Implementierung eines Aspekts zur Fadensynchronisation beim Zugriff auf Betriebsmittel des Systems.

9.1 Prozesszustände

9.1.1 Abstrakte und konkrete Zustände

Nach dem allgemeinen Verständnis im Kontext der Betriebssysteme wird unter dem Begriff Prozess die Ausführung eines Programms verstanden [47]. Ein Prozess entsteht durch die Anforderung an das Betriebssystem ein Programm zu starten, und endet, wenn das Programm entweder abgearbeitet ist oder während der Aus-

führung des Prozesses ein Abbruch nötig wird. Zwischen dem Entstehen und dem Beenden eines Prozesses kann dieser mehrere Zustände einnehmen. Abbildung 9.1 zeigt ein Zustandsdiagramm, das die wichtigsten Prozesszustände und Zustandsübergänge beschreibt. Obwohl Diagramme wie dieses in den meisten Lehrbüchern über Betriebssysteme (zum Beispiel [105, 98]) präsentiert werden, handelt es sich um eine grobe Vereinfachung. Das UNIX Prozesszustandsdiagramm [7] weist beispielsweise bereits neun verschiedene Zustände auf. Hier wird unter anderem noch unterschieden, ob ein Prozess im Benutzer- oder Systemmodus läuft und ob er eventuell auf ein externes Speichermedium ausgelagert wurde.

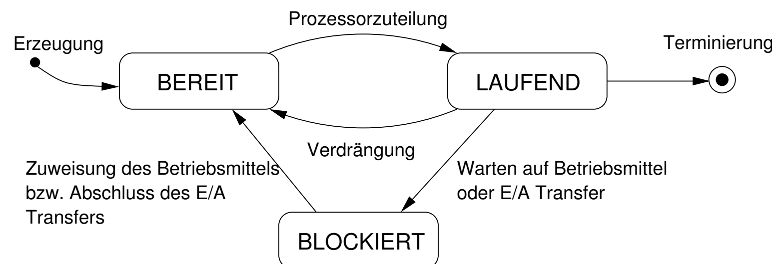


Abbildung 9.1: Die wichtigsten Prozesszustände

Tatsächlich könnte man in praktisch jedem Betriebssystem noch wesentlich mehr Prozesszustände unterscheiden. So wird ein Prozess mit Hilfe einer Datenstruktur, die in vielen Fällen als Prozesskontrollblock bezeichnet wird, durch das Betriebssystem verwaltet. Mindestens jede mögliche Belegung dieser Datenstruktur stellt einen unterscheidbaren Zustand dar. Damit ergeben sich Millionen von Möglichkeiten. Die in Zustandsdiagrammen dargestellten Prozesszustände abstrahieren also von unwichtigen Details und fassen konkrete Zustände zusammen. Daher soll fortan zwischen abstrakten und konkreten Prozesszuständen unterschieden werden.

Abstrakte Prozesszustände dienen nicht nur der Didaktik in Lehrbüchern. Sie werden tatsächlich in laufenden Systemen ermittelt und benutzt. So können sie “von außen” abgefragt werden und einem Systemadministrator einen Überblick über das System verschaffen. Auch intern können sie genutzt werden. Zum Beispiel kann mit abstrakten Zuständen wesentlich leichter überprüft werden, ob eine bestimmte Zustandsveränderung im gegenwärtigen Zustand erlaubt ist. Wenn ein Prozess entsprechend Abbildung 9.1 im Zustand BEREIT ist und versucht wird, diesen Prozess zu blockieren, muss beispielsweise ein Systemfehler vorliegen, da ein entsprechender Zustandsübergang nicht erlaubt ist. In der Regel ist es wesentlich leichter, den abstrakten Prozesszustand für diesen Test zu benutzen, als zu prüfen, ob der jeweilige Prozess in der Liste der rechenbereiten Prozesse enthalten ist.

Die folgenden Abschnitte beschreiben einen aspektorientierten Entwurf einer Prozessverwaltung, bei der im Sinne einer Programmfamilie abstrakte Prozesszu-

stände und die damit zusammenhängenden Systemeigenschaften als konfigurierbar angesehen werden. Dabei werden Feature-Diagramme, *Concern*-Hierarchien und Klassen/Aspekte-Diagramme zu Hilfe genommen. Nach jedem Entwurfsabschnitt folgt eine kurze Diskussion, um besondere Probleme oder Vorteile der aspektorientierten Herangehensweise hervorzuheben.

9.1.2 Abstrakte Prozesszustände in der Programmfamilie

Entwurf des Feature Modells

Sollen innerhalb eines Systems abstrakte Prozesszustände verwendet werden, sind während des Entwurfs folgende Fragen zu beantworten:

- Welche Zustände sollen unterschieden werden?
- Wie werden die Zustände ermittelt? Zum Beispiel:
  - Buchhaltung in einer Zustandsvariablen pro Prozess
  - Ermittlung des Zustands bei Bedarf anhand des konkreten Zustands
- Wann genau sollen die Zustandsübergänge stattfinden?
- Wie soll auf den Versuch eines ungültigen Zustandsübergangs reagiert werden?

Die Antworten auf diese Fragen wirken sich natürlich auf die Systemeigenschaften aus. So würde beispielsweise die Buchhaltung mit Hilfe einer Zustandsvariablen den Prozesskontrollblock vergrößern. In vielen eingebetten Systemen ist dies sehr kritisch, da gerade der dafür benötigte RAM Speicher dort oft knapp bemessen ist. Darüber hinaus würde dann bei jeder Prozessumschaltung Zeit für die Aktualisierung der Zustandsvariablen verbraucht werden. Auf der anderen Seite kann eine Ermittlung des Zustands bei Bedarf noch erheblich mehr Zeit erfordern. Welche Variante günstiger ist, hängt also vom Anwendungsprofil und von der Häufigkeit der internen Zustandsabfragen ab.

Im Kontext einer Programmfamilie bestimmt das Anwendungsprofil aufgrund der Anpassung Struktur und Verhalten des Systems. Damit sind auch interne Abfragen des abstrakten Zustands dem Anwendungsprofil unterworfen und Anworten auf die gestellten Fragen können zum Implementierungszeitpunkt nicht gegeben werden. Erst zum Konfigurierungszeitpunkt ist das nötige Wissen vorhanden, so dass auch der Code zur Ermittlung und Bereitstellung eines abstrakten Prozesszustandes konfigurierbar sein muss. Abbildung 9.2 auf der nächsten Seite zeigt, wie bei einem solchen Entwurf ein Feature-Diagramm aussehen könnte, das die verschiedenen möglichen Systemeigenschaften und ihre Beziehungen beschreibt.

Das Diagramm kann direkt aus der Fragenliste abgeleitet werden. Die Reaktion auf den Versuch eines unerlaubten Zustandsübergangs in Form einer Fehlerbehandlung ist optional, da dieser Fall im produktiven Einsatz nicht mehr auftreten sollte.

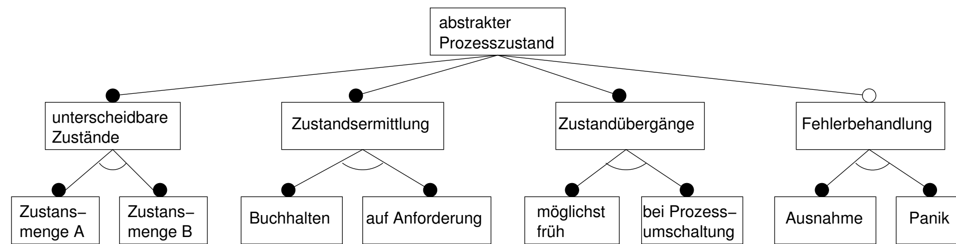


Abbildung 9.2: Feature-Diagramm für das Konzept des abstrakten Prozesszustands

Diskussion

Feature Modelle sind noch völlig unabhängig von der späteren Softwarestruktur oder der Frage, ob die damit beschriebenen Systemeigenschaften überhaupt realisierbar sind. Es spielt auch keine Rolle, ob eine bestimmte Eigenschaft ein *Crosscutting Concern* ist. Obwohl Feature Modelle zur Analyse einer Domäne gedacht sind, lassen sie sich sehr gut zur Spezifikation der Anforderungen an die Variabilität einer Programmfamilie verwenden. Für diese Fallstudie ist ein Feature Modell ein adäquates Beschreibungsmittel für die Anforderungen.

9.1.3 Verbindung von abstrakten und konkreten Zuständen

Grobentwurf der *Concern*-Hierarchie

Das in Abbildung 9.2 gezeigte Feature-Diagramm kann als Ergänzung zu dem Feature-Diagramm des Konzepts "Kontrollfaden" in Abbildung 5.1 auf Seite 52 angesehen werden. Es erläutert das Feature "Zustand" näher. Voraussetzung ist dabei, dass der Unterschied zwischen den Konzepten "Prozess" und "Faden" vernachlässigt wird. Dies soll hier der Einfachheit halber geschehen.

Als nächster Schritt im Entwurf muss nun eine *Concern*-Hierarchie gefunden werden. Sie beschreibt Funktionen mit Abhängigkeitsbeziehungen und *Crosscutting Concerns* mit Einwirkungsbeziehungen. Hilfreich ist dabei, dass häufig Features direkt als Funktionen in die *Concern*-Hierarchie übernommen werden können. Dies zeigte sich auch schon in Abbildung 5.3 auf Seite 55. Dort wurde beispielsweise aus einem Feature "Kommunikation" eine gleichnamige konfigurierbare Funktion. Die wesentliche neue Information in der *Concern*-Hierarchie besteht darin, dass die Funktion "Kommunikation" von der Funktion "Zustand" und diese wiederum von der Funktion "CPU Zuteilung" abhängt.

Berücksichtigt man die im letzten Abschnitt angestellten Überlegungen zu den vielen Möglichkeiten, mit dem abstrakten Prozesszustand umzugehen, fällt auf, dass die Einordnung des Zustands als Funktion zwischen "Kommunikation" und "CPU Zuteilung" wohl kaum den Anforderungen gerecht wird. Es handelt sich stattdessen um ein *Crosscutting Concern*, das auf allen Ebenen oberhalb von "Dispatching" wirkt bzw. je nach Konfigurierung wirken kann. "Dispatching" selbst

soll zustandslos arbeiten. Aufgrund dieser Überlegung ergibt sich die in Abbildung 9.3 dargestellte *Concern*-Hierarchie. Die Abhängigkeit der prozessverwaltenden Funktionen vom dem *Crosscutting Concern* existiert nur, wenn der abstrakte Zustand intern benutzt wird. Das *Crosscutting Concern* “abstrakter Prozesszustand” ist in sich entsprechend dem Feature-Diagramm in Abbildung 9.2 konfigurierbar. Ein verfeinerter Entwurf dieses Teils erfolgt in Abschnitt 9.1.4.

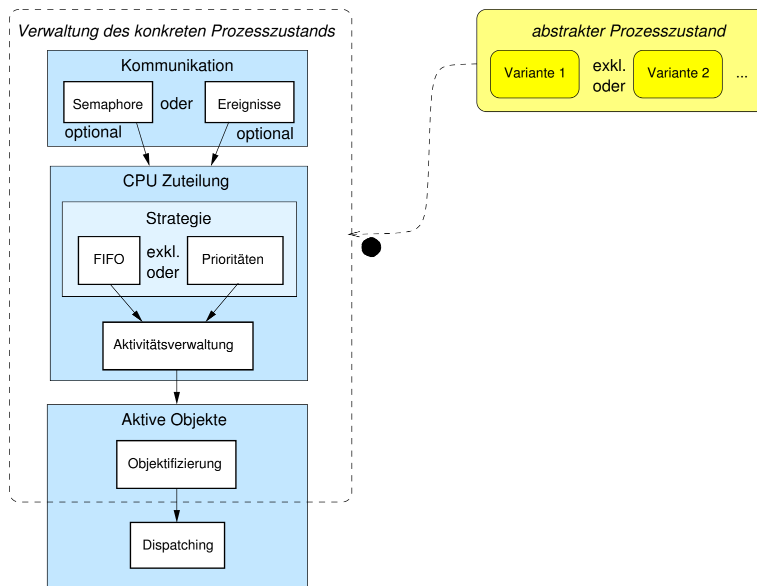


Abbildung 9.3: Beziehung zwischen konkreten und abstrakten Prozesszuständen in einer *Concern*-Hierarchie

Ein *Crosscutting Concern* wie der abstrakte Prozesszustand braucht ein relativ weitgehendes Wissen über die Funktionen, auf die er wirkt. Soll beispielsweise ein abstrakter Prozesszustand WAITING_EVENT erkannt werden, der aussagt, dass ein Prozess gerade auf ein Ereignis wartet, dann muss der Aspekt die Interna der Funktion “Ereignisse” oder mindestens die Schnittstellen und die damit verbundene Semantik kennen. Ersteres ist kritisch, da der aus dem *Crosscutting Concern* resultierende Programmcode dann bei Änderungen an “Ereignisse” angepasst werden müsste. Wartungsprobleme sind damit vorprogrammiert. Die zweite Variante ist dagegen akzeptabel, da die Kenntnis von Schnittstellen und der damit verbundenen Semantik nichts anderes ist als das Wissen, das auch höher in der Hierarchie angesiedelte herkömmliche Funktionen haben. Es kann beispielsweise davon ausgegangen werden, dass der WAITING_EVENT Zustand erreicht wird, wenn ein Prozess eine potentiell blockierende Primitive zum Warten auf ein Ereignis aufruft, die wiederum mit Hilfe der CPU Zuteilungsfunktion die Weitergabe der CPU an den nächsten rechenbereiten Prozess veranlasst. Wenn für beide Operationen im fertigen System der Name bekannt ist, kann ein Aspekt mit einer entsprechenden

Advice-Definition dieses Verhalten erkennen und darauf zum Beispiel durch Setzen einer Zustandsvariablen reagieren.

Diskussion

Bereits bei diesem Grobentwurf sind interessante Punkte zu bemerken. Das *Crosscutting Concern* verwendet zwar in diesem Beispiel ähnlich wie eine höher angesiedelte Funktion ein weitreichendes Wissen über die Funktionen, die den konkreten Prozesszustand verwalten. Dennoch hat der Entwickler im Vergleich zu einer Funktion nun einige zusätzliche Möglichkeiten zur Verfügung. Neben dem direkten Benutzen einer Funktion kann ein *Crosscutting Concern* Aktionen auslösen, wenn von anderer Seite aus eine bestimmte Funktion benutzt wird. Damit kann ein viel stärkeres Maß an Kontrolle ausgeübt werden, als es etwa mit einer herkömmlichen Funktion durch das Überdefinieren von Methoden innerhalb einer Vererbungshierarchie möglich ist.

Darüber hinaus ist ein wichtiges Unterscheidungsmerkmal, dass Funktionen, die andere Funktionen benutzen, diese zwingend benötigen. Fehlt in einer Konfiguration auch nur eine der benutzten Funktionen, so muss die benutzende Funktion ebenfalls entfallen. Ein *Crosscutting Concern* ist dagegen nicht grundsätzlich auf die Existenz aller Funktionen, auf die es einwirkt, angewiesen. Man muss also beispielsweise auf den abstrakten Prozesszustand nicht verzichten, nur weil in der gewünschten Systemkonfiguration keine Funktion “Ereignisse” benötigt wird. Das *Crosscutting Concern* wirkt in diesem Fall einfach auf alle Funktionen, die übrig sind.

9.1.4 Strukturierung des *Crosscutting Concerns* “abstrakter Prozesszustand”

Verfeinerung der *Concern*-Hierarchie

Unter Berücksichtigung des Feature-Diagramms in Abbildung 9.2 auf Seite 138 lässt sich das *Crosscutting Concern* “abstrakter Prozesszustand” noch weiter verfeinern. Ein guter Ansatzpunkt ist auch hier, jedes Feature als Funktion oder *Crosscutting Concern* anzusehen und diese dann zueinander in Beziehung zu setzen.

Abbildung 9.4 auf der nächsten Seite zeigt einen so entstandenen verfeinerten Entwurf. Es ist zu erkennen, dass sich die vier Features aus der ersten Ebene des Feature-Diagramms in der *Concern*-Hierarchie wiederfinden:

Unterscheidbare Zustände: Dieses Feature wird durch eine Funktion “Zustände” realisiert, die dafür verantwortlich ist, dass alle unterscheidbaren abstrakten Zustände als Werte eines Zustandsdatentyps dargestellt werden können und dass eventuell nötige Operationen auf diesem Datentyp zur Verfügung gestellt werden. Alle von dieser Funktion abhängenden *Concerns* müssen so konfiguriert werden, dass sie zu der Menge der unterscheidbaren Zu-

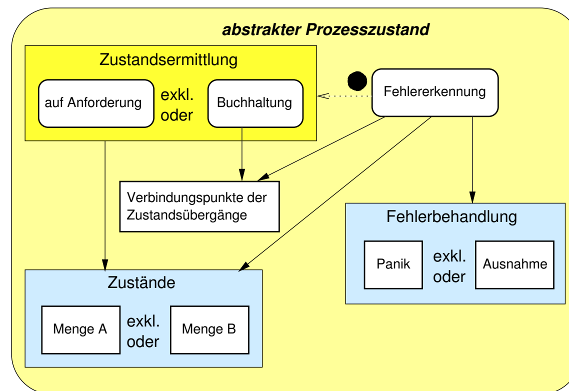


Abbildung 9.4: Verfeinerung des *Crosscutting Concerns* "abstrakter Prozesszustand"

stände passen. Die Menge der Zustände hängt wiederum davon ab, welche Prozesszustände intern oder von außen abgefragt werden sollen.

Zustandsermittlung: Die Zustandsermittlung mit ihren Unterfeatures wird direkt in ein konfigurierbares *Crosscutting Concern* abgebildet. Im Fall der Konfiguration "auf Anforderung" müssen Funktionen zur Prozesshierarchie hinzugefügt werden, die auf jeder Ebene in der Lage sind, anhand des konkreten Prozesszustandes den abstrakten Zustand zu ermitteln. Dabei spielt das Feature "Zustandsübergänge" mit hinein. Wenn zum Beispiel der Zustandsübergang eines Prozesses erst beim Prozesswechsel erfolgen soll, müsste für den aktuellen Prozess als Zustand LAUFEND geliefert werden, auch wenn dieser bereits auf der Warteliste der CPU Zuteilung steht und es ganz sicher zu einem Prozesswechsel kommen wird. Im Fall der "Buchhaltung" muss die Prozesshierarchie ergänzt werden, so dass beim Zustandswechsel eine Zustandsvariable pro Prozess entsprechend angepasst wird. Darüber hinaus ist eine Funktion in die Hierarchie einzufügen, mit der dieser Zustand abgefragt werden kann. Sowohl bei der "Buchhaltung" als auch bei der Zustandsermittlung "auf Anforderung" sollten die "Verbindungspunkte der Zustandsübergänge" in einer gesonderten Funktion gekapselt werden. In einer aspektorientierten Implementierung ist dies mit einer Liste von *Pointcut*-Definitionen möglich. Damit soll erreicht werden, dass Änderungen an den Funktionen der Prozesshierarchie sich nur auf eine Funktion auswirken.

Zustandsübergänge: Wie bei den Erläuterungen zu "Zustandsermittlung" bereits angesprochen wurde, spielt der Moment des Zustandsübergangs für alle Arten der Zustandsermittlung eine so wichtige Rolle, dass es nicht möglich ist, diese Eigenschaften als davon getrennte Funktion anzusehen. Stattdessen wird dieses Feature so behandelt als wäre es ein Unterfeature von "Zustandsermittlung". Damit muss beispielsweise die "Buchhaltung" so konfigurier-

bar sein, dass Zustandsübergänge wahlweise “möglichst früh” oder erst “bei Prozessumschaltung” erfasst werden.

Fehlerbehandlung: Das Feature “Fehlerbehandlung” lässt sich als konfigurierbare Funktion ansehen. Zusätzlich dazu existiert in der *Concern*-Hierarchie ein *Crosscutting Concern* “Fehlererkennung”. Die Fehlererkennung wirkt dort, wo Zustandsübergänge stattfinden und fragt mit Hilfe der Zustandsermittlung den vorherigen Zustand ab. Wenn der entsprechende Zustandsübergang nicht erlaubt ist, wird die Fehlerbehandlung für eine entsprechende Reaktion benutzt. Wenn Fehlererkennung und Zustandsermittlung an den gleichen Verbindungspunkten wirken, ist es wichtig, dass die Fehlererkennung zuerst arbeiten kann, damit die Zustandsabfrage noch den vorherigen Prozesszustand liefert. Durch die Fehlererkennung wird auch festgelegt, welche Fehler überhaupt erkannt werden sollen. Man kann dabei streng, weniger streng oder auch gar nicht kontrollieren. Hierbei handelt es sich um Systemeigenschaften, die zu konfigurieren für einen Benutzer durchaus interessant sein könnte, so dass das Feature Modell erweitert werden müsste.

Diskussion

Bei der Verfeinerung des *Crosscutting Concerns* “abstrakter Prozesszustand” hat sich gezeigt, dass dieser wiederum aus *Crosscutting Concerns* und herkömmlichen Funktionen besteht. Es ist typisch, dass bei einer fortschreitenden Verfeinerung nur noch wenige minimale *Crosscutting Concerns* übrig bleiben, deren Aufgabe nur darin besteht, Funktionen auf der Zielseite des Einwirkens mit direkt benutzten Funktionen zu verbinden. Es handelt sich also um Bindeglieder, die sehr flexibel Funktionen verknüpfen können.

Ein besonderes Augenmerk verdient an dieser Stelle noch die “Zustandsermittlung” “auf Anforderung”. Dabei muss der konkrete Prozesszustand interpretiert werden. Damit dies nicht Detailkenntnisse der internen Datenstrukturen der Prozessverwaltung erfordert, müssen auf jeder Ebene entsprechende Zugriffsfunktionen angeboten werden. Beispielsweise sollte auf der Ebene der CPU Zuteilung eine Methode `is_ready()` existieren, mit der abgefragt werden kann, ob auf dieser Ebene ein bestimmter Prozess auf die CPU Zuteilung wartet. Auf diese Weise werden in der Prozess-Hierarchie keine Entwurfsentscheidungen vorweggenommen und die Zustandsermittlung basiert lediglich auf einer Schnittstellendefinition und nicht auf internen Details. An diesem Beispiel ist zu erkennen, dass Komponentencode für das Einwirken von Aspekten entworfen sein sollte (“Design for Aspect Intervention” [77]). Dies erfordert allerdings Disziplin und birgt das Risiko, dass bei mangelner Sorgfalt Aspekte zu stark von bestimmten Verbindungspunkten im Komponentencode abhängen, wodurch die Wartbarkeit beeinträchtigt wird.

9.1.5 Ergebnis des Entwurfsprozesses

Entwurf der Klassen/Aspekt-Struktur

Nach dem verfeinerten Entwurf der *Concern*-Hierarchie folgt nun der Übergang zur Modulstruktur, die in diesem Fall aus Aspekten und Klassen bestehen soll. Aspekte werden zur Implementierung von *Crosscutting Concerns* und Klassen zur Implementierung von Funktionen genutzt. Damit ergibt sich die in Abbildung 9.5 dargestellte Klassen/Aspekt-Struktur. Durch die zugeordneten Codefragmente wird gleichzeitig angedeutet, wie der fertige Programmcode aussehen könnte. Interessant ist dabei zu bemerken, dass `ProcState` und `TransitionPoints`, die die Funktionen "Zustände" und "Verbindungspunkte der Zustandsübergänge" realisieren, nicht konfigurierbar ausgelegt sind. `ProcState` ist beispielweise ein Aufzählungstyp, der sowohl detallierte Zustände wie WAITING_EVENT und WAITING_COUNTER als auch zusammengefasste Zustände wie WAITING umfasst. Dies ist mit keinerlei zusätzlichem Ressourcenverbrauch verbunden und erlaubt gleichzeitig den Einsatz in allen Anwendungsszenarien.

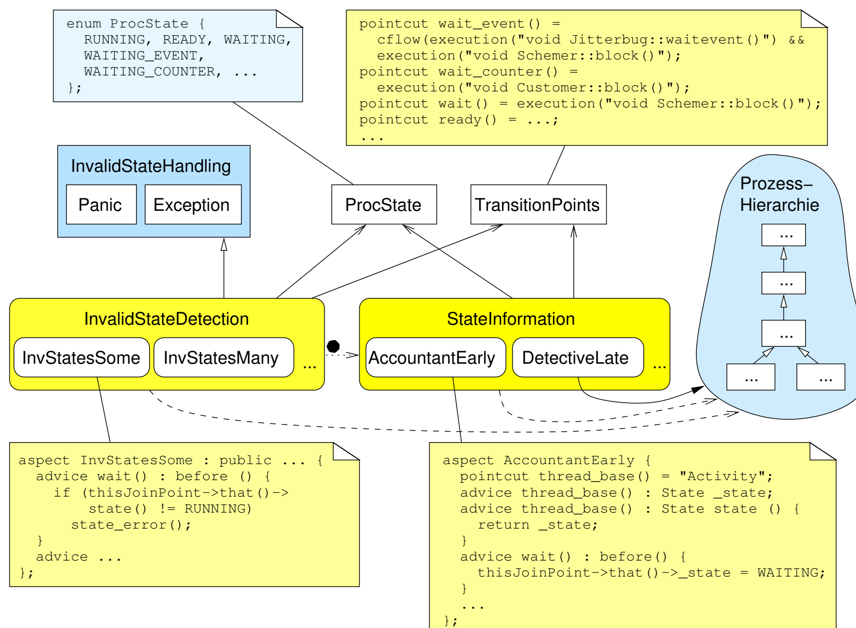


Abbildung 9.5: Klassen/Aspekt-Struktur des "abstrakten Prozesszustands"

Als Beispiel für einen Zustandsermittler wurde in der Abbildung der Quellcode für den Aspekt `AccoutantEarly` aufgeführt. Er fügt zur Prozessbasisklasse `Activity` eine Zustandsvariable zwecks Buchhaltung und eine Abfragefunktion hinzu, die lediglich den Inhalt der Zustandsvariablen zurückliefert. Das Setzen der Zustandsvariablen erfolgt mit Hilfe von *Advice*-Code, der entsprechend der Definitionen in `TransitionPoints` aktiviert wird.

Die Erkennung unerlaubter Zustandsübergänge benutzt ebenfalls die Definitionen in `TransitionPoints`. Wichtig ist für den hier ausgewählten Aspekt, dass seine Aktivierung vor der Aktivierung der Zustandsermittlung, also einem `StateInformation` Aspekt, erfolgt. Auf diese Weise kann der bisherige Prozesszustand mit Hilfe der Methode `state()` abgefragt werden. Durch die Benutzung der `state()` Methode des Prozessobjekts ist die Fehlererkennung unabhängig von der Konfiguration des `StateInformation` Aspekts. Die im Fehlerfall aufgerufene Funktion `state_error()` stammt aus der konfigurierbaren Basisklasse `InvalidStateHandling`.

Diskussion

Auf den ersten Blick mag das hier erzielte Ergebnis reichlich kompliziert wirken. Dabei ist jedoch zu berücksichtigen, dass ein großes Maß an Variabilität einer Systemeigenschaft erreicht wurde, die bei anderen Systemen absolut unveränderlich ist. Es wurde zudem erreicht, dass die Zahl der Konfigurationspunkte in der Implementierung (Abbildung 9.5 auf der vorherigen Seite) nicht größer ist als die Zahl der Eigenschaften im Feature Modell (Abbildung 9.2 auf Seite 138). So muss lediglich die gewünschte Ausprägung von `InvalidStateHandling`, `InvalidStateDetection` und `StateInformation` gewählt werden. Ohne den Einsatz von Aspekten wäre dies kaum möglich gewesen.

9.1.6 Bewertung

Bezieht man diese erste Fallstudie auf die Bewertungskriterien, so kann festgehalten werden, dass *Concern*-Hierarchien als Modellerweiterung zu funktionalen Hierarchien hier durchaus geeignet waren, um eine bestimmte Systemeigenschaft konfigurierbar in eine Betriebssystemfamilie einzubringen. Dabei wurde als Zahl der Konfigurierungspunkte das Minimum, also die Zahl der Eigenschaften im Feature Modell, erreicht. Bei dieser Bewertung muss allerdings bedacht werden, dass es sich hier um ein kleines Beispiel mit nur einer Verfeinerungsstufe handelt, so dass ein Bedarf für weitere Konkretisierungen der Modellbeschreibung nicht auszuschließen ist.

Crosscutting Concerns haben sich auf der Entwurfsebene als mächtiges und hilfreiches Konzept dargestellt. Gleichzeitig wurde aber auch deutlich, dass dem Entwickler Disziplin abverlangt wird, um Wartungsprobleme zu vermeiden, die auftreten können, wenn Aspekte zuviel Wissen über die Interna bestimmter Module nutzen. Außerdem wurde in diesem Beispiel von einer notwendigen Ordnungsbeziehung Gebrauch gemacht, obwohl solche Beziehungen bereits in Abschnitt 5.1.6 auf Seite 60 als kritisch beurteilt wurden. Das Zurückgreifen auf diese Beziehungsart war in dem Beispiel notwendig, um die Erkennung unerlaubter Zustandsübergänge von der Zustandsermittlung zu trennen.

9.2 Unterbrechungssynchronisation

In dieser Fallstudie wird die Unterbrechungssynchronisation in der PURE Betriebssystemfamilie betrachtet. Unterbrechungssynchronsation ist eine der zentralen Aufgaben von Betriebssystemen. Abbildung 9.6 verdeutlicht, wozu sie nötig ist. Dort ist zu sehen, dass der Betriebssystemkern Datenstrukturen verwaltet, die sowohl aus dem normalen Kontrollfluss heraus als auch von Unterbrechungsbehandlungsroutinen benutzt werden. Ein Beispiel dafür ist die Warteliste mit rechenbereiten Kontrollfäden für die CPU Zuteilung.

Ein Problem kann nun auftreten, wenn sich zum Beispiel zwei Fäden über einen Semaphor synchronisieren und der eine den anderen durch eine `v()` bzw. `signal()` Operation weckt. Der geweckte Faden wird dann in die CPU Warteliste eingereiht. Da Unterbrechungen durch externe Hardware ausgelöst werden, kann dieser Vorgang jederzeit unterbrochen werden. Es wird dann eine Behandlungsroutine aktiviert, die beispielsweise dafür verantwortlich ist, Daten von einem Peripheriegerät abzuholen und an einen darauf wartenden Faden auszuliefern. Im Rahmen dieser Auslieferung wird der wartende Faden geweckt, was bedeutet, dass auch er in die CPU Warteliste eingereiht wird. Damit greifen der synchrone Kontrollfluss und die asynchrone Unterbrechungsbehandlung quasi gleichzeitig auf die Warteliste zu, was ohne Synchronisation leicht zu einer Zerstörung der Datenstruktur führen kann. Der Programmcode zur Manipulation der CPU Warteliste wird daher auch als “kritischer Abschnitt” eingestuft.

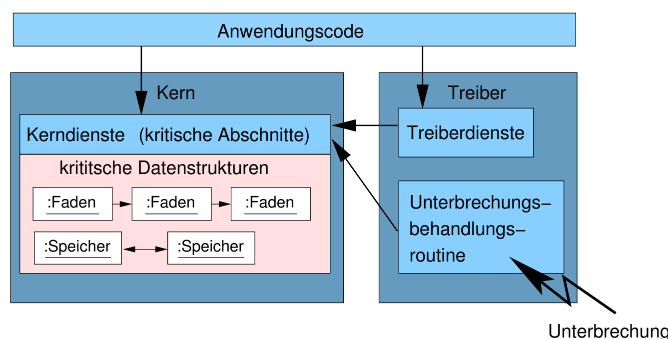


Abbildung 9.6: Kritische Abschnitte im Betriebssystemkern

Die Art der Unterbrechungssynchronisation hat direkte Auswirkungen auf die Systemleistung und das Antwortverhalten. Die einfachste Form zu synchronisieren, besteht in dem Verbieten der Unterbrechungsbehandlung über ein entsprechendes CPU Steuerregister. Kürzere Reaktionszeiten und eine verminderte Gefahr, dass Unterbrechungen verloren gehen, sind mit dem Pro-/Epilogmodell von PURE möglich [97]. Dabei kann das hardwaremäßige Sperren von Unterbrechungen komplett vermieden werden [96].

Kritischer als die Auswahl der Synchronisationsprimitiven ist für den Ent-

wurf eines Betriebssystems allerdings die Festlegung einer Strategie zur Unterbrechungssynchronisation. So kann zum Beispiel grob- oder feingranular gesperrt oder eine Kompromisslösung angestrebt werden. Um die Strategie auf der Implementierungsebene umzusetzen, müssen entsprechend der Strategie Punkte im Systemcode identifiziert werden, an denen Aufrufe bestimmter Sperrprimitiven eingefügt werden. Beim feingranularen Sperren befinden sich diese Punkte sehr dicht an den kritischen Abschnitten, während sie beim grobgranularen Sperren sehr weit davon entfernt sein können. Dadurch, dass sehr viele Subsysteme von Sperraufrufen durchzogen sind, sind Änderungen an dieser Strategie aufwändig und wegen der notwendigen manuellen Eingriffe riskant.

Wie anhand dieser Beschreibung zu erkennen ist, handelt es sich bei der Strategie zur Unterbrechungssynchronisation um ein *Crosscutting Concern*. Es ist daher ein gutes Beispiel um zu zeigen, ob und welche Vorteile die Anwendung von AOP im Betriebssystemumfeld bringt. Die folgenden Abschnitte beschreiben dazu zunächst den Status quo der Unterbrechungssynchronisation bei PURE. Dann werden die Probleme dieser Implementierung analysiert und zwei aspektorientierte Implementierungsvarianten vorgestellt. Somit ist ein direkter Vergleich einer aspektorientierten Lösung mit einer älteren nicht-aspektorientierten Lösung möglich.

9.2.1 Der Status quo

Bei PURE muss vor einem kritischen Codeabschnitt die Synchronisationsprimitive `lock.enter()` und hinterher `lock.leave()` aufgerufen werden. Sollten dazwischen Unterbrechungen auftreten, werden nach dem Pro-/Epilog-Modell die damit verbundenen kritischen Operationen solange verzögert, bis der kritische Abschnitt abgearbeitet ist. Eine Analyse der Quelltexte zeigt, dass es 166 dieser Aufrufe in PURE gibt, die sich auf 15 Klassen verteilen. Abbildung 9.7 auf der nächsten Seite zeigt einen vereinfachten Ausschnitt aus der Klassenhierarchie von PURE, die 8 dieser 15 Klassen enthält. Sie sind durch eine dunklere Unterlegung markiert. Hier zeigt sich, wie viele verschiedene Klassen in unterschiedlichen Subsystemen (*Osek*, *Thread* und *Case*) von der Unterbrechungssynchronisation betroffen sind. Jede dieser Klassen enthält im Durchschnitt etwa 11 entsprechende Aufrufe.

Das dargestellte Klassendiagramm spiegelt den aus Schichten bestehenden Entwurf von PURE wieder, wobei jede (oftmals konfigurierbare) Schicht als neue Ebene im Vererbungsbaum implementiert ist. Da die Aufrufe der Synchronisationsprimitive beim Entwurf als eigene Schicht betrachtet wurden, liegen sie in der Regel auch in eigenen Klassen vor. Innerhalb solcher Synchronisationsklassen (zum Beispiel `Monitor`) werden alle Methoden der Basisklasse durch neue Implementierungen überdeckt, die lediglich den Aufruf an die überdeckte Methode weiterleiten, vorher jedoch `lock.enter()` und hinterher `lock.leave()` aufrufen.

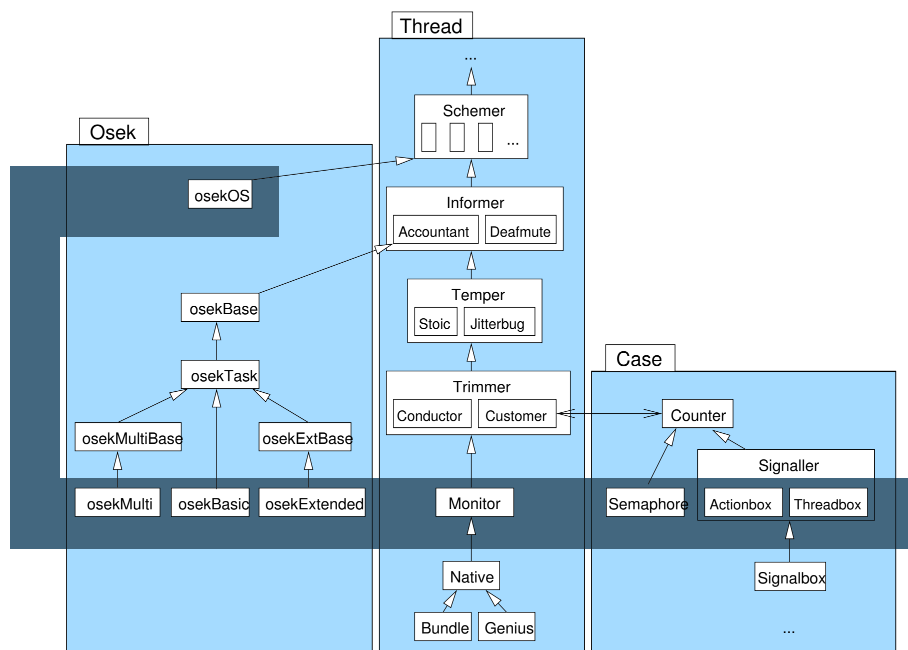


Abbildung 9.7: Zugriffe auf die globale Sperrvariable in PURE

9.2.2 Problemanalyse

Das häufige Aufrufen derselben Funktionen bedeutet zunächst lediglich, dass ein hohes Maß an Wiederverwendung vorliegt. Im Fall der Unterbrechungssynchronisation von PURE muss jedoch beachtet werden, dass es aufgrund der Konfigurierbarkeit zu verschiedenen Problemen kommt:

Wiederverwendbarkeit

Viele Subsysteme (*Osek*, *Thread* und *Case* in Abbildung 9.7) hängen von der Implementierung, oder wenigstens von der Schnittstelle der Synchronisationsprimitiven ab. Daher können sie in einem anderen Kontext oder in einer anderen Systemkonfiguration nicht ohne fehlerträchtige Codeänderungen wiederverwendet werden, sofern dort eine andere (oder keine) Form der Unterbrechungssynchronisation benutzt wird. Das Ziel, Systeme aus wiederverwendbaren Komponenten aufzubauen, ist so schwer erreichbar.

Bei PURE tritt der Fall auf, dass in bestimmten Systemkonfigurationen auf Unterbrechungssynchronisation verzichtet werden kann. Um Ressourcen zu sparen, sollten in diesem Fall Zugriffe auf die globale Sperrvariable `lock` durch Aufrufe der Methoden `enter()` und `leave()` nicht stattfinden. In der bisherigen Implementierung von PURE wurde dieses Problem dadurch gelöst, dass die globale Sperrvariable `lock` eine Instanz der Klasse `Sluice` ist, die wiederum per Konfigurations-

schalter auf `Guard` oder `VoidGuard` abgebildet wird. `Guard` enthält die eigentliche Implementierung der Sperroperationen, während `VoidGuard` diese Methoden zwar bereitstellt, ihre Implementierung aber leer lässt. Diese Lösung benötigt lediglich einen Konfigurationspunkt, ist aber trotzdem nicht als optimal zu erachten. Letztlich bedeutet dieser Ansatz, dass für Klassen, die optionale Funktionen erbringen, jeweils leere ("Dummy") Klassen bereitgestellt werden müssen, damit das System in allen Konfigurationen übersetzbar ist. Eine Suche im Quelltext zeigt tatsächlich, das noch weitere solcher Dummy-Klassen existieren. Besser wäre es, wenn der Entwickler der Klasse `Guard` sich nicht um diese Abhängigkeit von Systemkonfigurationen kümmern müsste.

Beschränkung der Schnittstelle

Ein zweites Problem besteht darin, dass die Schicht, in der der Synchronisationscode hinzugefügt wird, automatisch eine Systemschnittstelle für Anwendungsentwickler definiert. Während die Klassen dieser Schicht oder davon abgeleitete Klassen benutzt werden dürfen, ist es nicht gestattet, Basisklassen davon einzusetzen, da dann das Synchronisationsverhalten durch die Anwendung beeinträchtigt werden könnte. Um dies doch zu tun, müsste ein Anwendungsprogramm selber für Synchronisation sorgen. Die Semantik ist auf diese Weise verwirrend uneinheitlich.

Probleme bei der Systemerweiterung

Ein drittes Problem stellt sich für Systementwickler. Erweiterungen der Synchronisationsklassen in Form abgeleiteter Klassen sind zunächst ungeschützt vor Unterbrechungen. Wenn dies doch erforderlich sein sollte, weil auch hier kritische Datenstrukturen verwaltet werden sollen, so muss wieder die Sperrvariable gesetzt werden. Allerdings muss dann sichergestellt werden, dass die Sperre wieder aufgehoben wird, bevor Funktionen der Basisklasse benutzt werden, da ein mehrfaches Sperren zu Fehlverhalten führt. Dieses kurzeitige Freigeben gibt jedoch eine Gelegenheit für Unterbrechungen, die in vielen Fällen nicht gestattet werden dürfen. Durch dieses Problem ist es für Entwickler ausgesprochen schwierig, Erweiterungen vorzunehmen, die kritische Datenstrukturen verwalten. Die bisherige Lösung besteht darin, Erweiterungen nur von Klassen vorzunehmen, die sich innerhalb des geschützten Systembereichs befinden. Die neue abgeleitete Klasse muss dann nochmals erweitert werden, um auch davon eine synchronisierte Variante zu haben. Auf diese Weise entstehen immer mehr Synchronisationsklassen.

Anpassungsfähigkeit an Änderungen der Strategie

Alle hier aufgeführten Probleme der existierenden Implementierung zeigen, dass es schwierig ist, Punkte für die Unterbrechungssynchronisation – sofern sie notwendig ist – festzumachen. Dies sollte auch nicht dem Entwickler einer einzelnen

Systemkomponente oder einem Anwendungsentwickler zugemutet werden. Stattdessen sollte beachtet werden, dass die Positionierung der Sperren für Unterbrechungen und die Art ihrer Implementierung strategische Entscheidungen sind. Sie kann große Auswirkungen auf das Gesamtverhalten eines Betriebssystems haben [49].

Im Sinne des Prinzips *Separation of Concerns* ist es vernünftig, die Strategieimplementierung von den einzelnen Systemkomponenten zu trennen. Erst dadurch wird es möglich, auch globale Systemanforderungen zu beachten, die der Entwickler einer einzelnen Komponente (zum Beispiel eines Gerätetreibers) nicht kennt und auch nicht kennen sollte. Die geforderte Zeit für die Aktivierung eines Fadens aus einer Unterbrechungsbehandlung heraus ist ein gutes Beispiel für solche globalen Anforderungen. Mit einer feingranularen Sperrstrategie lässt sich die Zeit klein halten. Dafür treten allerdings im Normalbetrieb durch das häufige Sperren und Freigeben Leistungsnachteile auf. Eine grobgranulare Sperrstrategie bewirkt das Gegenteil. Da beide Varianten je nach Systemkonfiguration ihre Vor- und Nachteile haben könnten, wäre es im Sinne der Philosophie von PURE, diese Systemeigenschaft in Form eines Features konfigurierbar zu gestalten. Bei der derzeitigen Art der Implementierung ist dies jedoch unmöglich.

9.2.3 Variante 1

Mit Hilfe eines Aspekts lässt sich die Implementierung der Sperrstrategie vom Komponentencode, also den Klassen der betroffenen Subsysteme, trennen. Bei der in Abbildung 9.8 auf der nächsten Seite gezeigten Implementierungsvariante 1 erfolgt dementsprechend der Aufruf von `lock.enter()` und `lock.leave()` nur noch innerhalb von *Advice*-Code[1]. Die Aufrufe innerhalb der bisherigen Synchronisationsschicht konnten entfernt werden. Stattdessen werden die zu dieser Schicht gehörenden Klassen mit Hilfe eines *Pointcuts* `layer()` innerhalb des Aspekts beschrieben. Der *Pointcut* `locked()` benutzt diese Definition und beschreibt alle Funktionen bei deren Betreten `lock.enter()` und bei deren Verlassen `lock.leave()` ausgeführt werden muss. Der zugewiesene *Pointcut*-Ausdruck für `locked()` beschreibt zunächst die Menge aller Funktionen aus der Synchronisationsschicht und zieht dann durch Schnittmengenbildung eine Reihe von Ausnahmen ab, auf die später noch eingegangen wird. Der Advice-Code für `locked()` ist am Ende der Aspektdefinition zu sehen.

Der Pointcut `upcall()` beschreibt alle virtuellen Funktionsaufrufe, bei denen der geschützte Bereich verlassen werden soll, da dadurch Anwendungscode ausgeführt wird. Dies ist bei PURE der Fall, wenn innerhalb der Klasse `Actionbox` die Funktion `Action::action()` aufgerufen wird. Der *Advice*-Code für `upcall()` verhält sich genau umgekehrt wie der für `locked()`. Vor dem Aufruf werden durch `lock.leave()` Epiloge zugelassen und nachher werden sie wieder gesperrt.

[1] Die hier gezeigten Quelltexte sind, um die Verständlichkeit zu erhöhen, gekürzt wiedergegeben. Dabei wurden Klassen und Funktionen ignoriert, die nicht zu den Subsystemen *Osek*, *Thread* und *Case* gehören (siehe Abbildung 9.7 auf Seite 147).

Die Pointcuts `enter()` und `leave()` definieren weitere Ein- und Austrittspunkte in den bzw. aus dem geschützten Kernbereich. Beispielsweise wird die Funktion `liedown()` in PURE aufgerufen, wenn kein rechenbereiter Faden zur Verfügung steht, so dass die CPU nutzlos warten muss. Da das System diesen Zustand nur wieder verlassen kann, wenn im Rahmen einer Unterbrechungsbehandlung ein Faden rechenbereit wird, muss im Rahmen der Ausführung von `liedown()` die Sperrprimitive `lock.leave()` aufgerufen werden.

```
aspect IntSync1 {
  // Synchronisationsschicht
  pointcut layer()  = within("osekExtended"||"osekBasic"||"osekMulti"||
    "Monitor"||"Actionbox"||"Threadbox"||"Semaphore");

  // Aufrufe, bei denen der geschützte Bereich verlassen wird
  pointcut upcall() = within("Actionbox") && call  ("% Action::action()");

  // Ein- und Austritte in den bzw. aus dem geschützen Bereich
  pointcut enter()  =
    (within(derived("Schemer")) && execution("% %::getup()")) ||
     execution("% osekOS::start(...)");
  pointcut leave()  = within(derived("Schemer")) &&
    (execution("% %::hello()") || execution("% %::liedown()"));

  // Zu schützende Funktionen
  pointcut locked() = (layer() && execution("% %::%(...)")) &&
                      !(enter() || leave()) &&
                      !execution("% Actionbox::unload()") &&
                      !execution("% Semaphore::p()") &&
                      !execution("% Semaphore::v()");

  // Advice-Code zum Sperren und Erlauben von Epilogen
  advice upcall() : before() { lock.leave (); }
  advice upcall() : after()  { lock.enter (); }
  advice locked() || enter() : before() { lock.enter (); }
  advice locked() || leave() : after()  { lock.leave (); }
};
```

Abbildung 9.8: Variante 1 des Aspekts zur Unterbrechungssynchronisation

Das Problem an dieser an sich funktionierenden Implementierung ist, dass der Aspekt Wissen über die Implementierung einzelner Klassen verwendet. Dies zeigt sich zum Beispiel an der Ausnahmebehandlung für die Funktionen `Semaphore::p()` und `v()`. Diese Funktionen tun nichts weiter, als die Funtionen `wait()` und `signal()` aufzurufen, die die eigentliche Funktionalität bereitstellen. Damit kann sich ein Benutzer den ihm angenehmeren Namen aussuchen. Da die Synchronisationsprimitiven `lock.enter()` und `lock.leave()` nicht geschachtelt benutzt werden dürfen, darf der Aspekt `p()` und `v()` nicht synchronisiert ausführen. Durch diese Maßnahme ist der Aspekt allerdings nicht robust in Bezug auf Änderungen am Komponentencode. Wenn beispielsweise ein Entwickler die Klasse `Semaphore` erweitert, so dass nun analog zu `p()` und `v()` auch noch die Namen `down()` und `up()` zugelassen werden, muss der Aspekt unbedingt angepasst werden,

da es sonst zu einem fehlerhaften Synchronisationsverhalten käme.

9.2.4 Variante 2

Die zweite Implementierungsvariante vermeidet die Schwäche der ersten, indem das Abstraktionsniveau auf die Subsystemebene angehoben wird. Der entsprechende Quelltext des Aspekts ist in Abbildung 9.9 auf der nächsten Seite zu sehen. Er besteht aus zwei Teilen: Einer Beschreibung der Klassen, die zu den verschiedenen PURE Subsystemen gehören, mit Hilfe von *Pointcut*-Definitionen und dem Aspekt selbst. Der Aspekt benutzt die Subsystemdefinitionen, um damit einen *Pointcut* namens `kernel()` zu definieren, der die zu synchronisierende Region beschreibt. Damit ist gemeint, dass die Ausführung von Epilogen unterbleiben muss, solange eine Methode einer der hier erfassten Klassen ausgeführt wird. Um dies zu gewährleisten, soll `lock.enter()` aufgerufen werden, wenn von außerhalb eine Funktion in dieser Region aufgerufen wird, und `lock.leave()`, wenn die Ausführung abgeschlossen ist und die Region verlassen wird. Die dafür relevanten Verbindungspunkte werden durch den Pointcut

```
pointcut locked() = call(kernel()) && !within(kernel());
```

beschrieben. Mit Hilfe des *Advice*-Codes für diesen *Pointcut* wird die Synchronisation beim Eintreten in die zu synchronisierende Region sichergestellt. Sprünge innerhalb der synchronisierten Region nach außen werden durch den *Pointcut* `unlocked()` erfasst. Hier handelt es sich um wenige Fälle, wie dem Aufruf der virtuellen Methode `Triplet::action()`, um einen neu erzeugten Faden zu starten. Der Advice-Code für `unlocked()` sorgt dafür, dass vor dem Verlassen der synchronisierten Region die Epiloge zugelassen und beim Wiedereintritt gesperrt werden.

Auch in dieser zweiten Variante kann die Nennung bestimmter Namen von Funktionen oder Klassen des Komponentencodes nicht vermieden werden. Der Vorteil gegenüber der ersten Variante ist jedoch, dass es sehr viel weniger sind und vor allem, dass kein Wissen über die Implementierung bestimmter Funktionen genutzt wird. Stattdessen werden hier lediglich Klassennamen, oder wie im Fall von `Triplet::action()`, dokumentierte Schnittstellen gebraucht. Die Menge der notwendigen Namen für die Beschreibung der Subsysteme ließe sich sogar noch weiter reduzieren, wenn bei PURE Gebrauch von C++ Namensräumen gemacht würde.

9.2.5 Vergleich

Tabelle 9.1 auf Seite 153 zeigt einen Vergleich der Codegrößen der alten Implementierung ("Status quo"), Variante 1 und Variante 2. Da PURE Systeme anwendungsspezifisch konfiguriert und zusammen mit der Anwendung gebunden werden, hängen Größen sehr stark von der jeweiligen Anwendung ab. Daher wurden für den Vergleich drei Testanwendungen gemessen: Die Anwendung **friend** benutzt eine kooperative CPU Zuteilung und keine Unterbrechungen, während **philo**

```
// Beschreibung der PURE Subsysteme
pointcut osek()   = derived("osekBase") || "osekOS";
pointcut thread() = derived("Schemer") && !osek();
pointcut case()   = derived("Counter");
// ...

// Aspekt zur grobgranularen Unterbrechungssynchronisation
aspect IntSync2 {
  // Zu synchronisierende Region
  pointcut kernel() = osek() || thread() || case() /* || ... */;

  // Eintritte in die zu synchronisierende Region
  pointcut locked() = call(kernel()) && !within(kernel());

  // Austritte aus der zu synchronisierende Region
  pointcut unlocked() = within(kernel()) &&
    (call("void Triplet::action()") /* || ... */);

  // Advice-Code zum Sperren und Erlauben von Epilogen
  advice unlocked() : before() { lock.leave (); }
  advice unlocked() : after()  { lock.enter (); }
  advice locked()   : before() { lock.enter (); }
  advice locked()   : after()  { lock.leave (); }
};
```

Abbildung 9.9: Variante 2 des Aspekts zur Unterbrechungssynchronisation

und **clock** eine unterbrechungsgesteuerte präemptive CPU Zuteilung benutzen. Die bei **philo** arbeitenden Fäden basieren auf der Klasse `Native` und Fäden bei **clock** basieren auf den Fadenabstraktionen des *Osek*-Subsystems (siehe Abbildung 9.7 auf Seite 147).

Wie die Messungen zeigen, ist mit der aspektorientierten Implementierung in keinem Fall ein nennenswert erhöhter Speicherplatzverbrauch verbunden. Bei der Implementierungsvariante 2 ist sogar eine Verringerung festzustellen. Sie rührt daher, dass hier Klassen, die in der Status quo Variante ausschließlich der Unterbrechungssynchronisation dienen, eliminiert wurden. Der Synchronisationscode wird dafür auf der Aufrufseite integriert. Dies führt zu einem kompakteren System, da eine Ebene in der Vererbungshierarchie, bei der wie hier virtuelle Funktionstabellen generiert werden müssen, vergleichsweise teuer ist. Hinzu kommt noch, dass bei Variante 2 auf die virtuellen Funktionen `liedown()` und `getup()` verzichtet wurde, was sich auf allen Ebenen der Vererbungshierarchie mit Speicherplatzersparnissen bemerkbar macht. Die Bedeutung dieser Funktionen wurde bereits in Abschnitt 9.2.3 angesprochen. Durch sie bekommt in der Status quo Variante die Synchronisationsschicht die Möglichkeit, den Epilog der Unterbrechungsbehandlung zuzulassen, bevor die CPU mangels rechenbereiter Fäden in einen Wartezustand übergeht. Dies ist ein typisches Beispiel für eine Situation, wo eine virtuelle Funktion nicht wegen der Möglichkeit des dynamischen Bindens gewählt wurde, sondern lediglich existiert, um die Verantwortlichkeiten der Schichten des Systems

Anwendung	**System**	**Größe** (in Bytes)			
		text	data	bss	Summe
friend	Status quo	4319	612	80	5011
	Variante 1	4575	612	80	5267
	Variante 2	3855	540	80	4475
philo	Status quo	8438	984	108	9530
	Variante 1	8518	984	108	9610
	Variante 2	8294	952	108	9354
clock	Status quo	9891	1160	768	11819
	Variante 1	9587	1144	772	11503
	Variante 2	9491	1128	736	11355

Tabelle 9.1: Vergleich des Speicherverbrauchs

zu trennen. Mit Hilfe eines Aspekts kann dasselbe Ergebnis ressourcensparender erreicht werden. Mit einer *Advice*-Definition für die Funktion, in der die CPU auf rechenbereite Fäden wartet, kann das gleiche Synchronisationsverhalten erreicht werden. Der Unterschied ist, dass durch das Weben die Aufrufe der Synchronisationsprimitiven direkt von der Wartefunktion aus aufgerufen werden. Dabei bleibt auf der Quelltextebene die Trennung der Zuständigkeiten erhalten.

Neben der Speicherplatzersparnis konnte bei dieser Fallstudie festgestellt werden, dass sich die Konfigurierung des Systems bei Variante 2 erheblich vereinfacht hat. So kann nun auf fünf Klassenaliase, ein generiertes Präprozessormakro und eine `#ifdef` Direktive verzichtet werden. Darüber hinaus konnten eine Dummy-Klasse sowie vier Klassen aus der Synchronisationsschicht gelöscht werden.

9.2.6 Bewertung

Diese Fallstudie hat am Beispiel der Unterbrechungssynchronisation gezeigt, dass tatsächlich globale Strategien eines Betriebssystems mit Hilfe von Aspekten modular und damit konfigurierbar implementiert werden können. Dies ist nicht zwangsläufig mit einem erhöhten Ressourcenverbrauch verbunden. Tatsächlich konnte der Speicherplatzverbrauch bei Variante 2 sogar leicht reduziert werden. Die Anzahl der Konfigurierungspunkte hat sich in diesem Beispiel wieder auf ein Minimum reduziert. Zahlreiche Klassenaliase konnten in diesem Zusammenhang gelöscht werden. Übrig ist nur die Aktivierung/Deaktivierung des Unterbrechungssynchronisationsaspekts bzw. die Auswahl einer Aspektvariante, sobald das im Feature Modell vorgesehen und implementiert ist.

Auch die Menge an Quellcode konnte reduziert werden. Aus den 166 anfangs erwähnten Aufrufen von Synchronisationprimitiven wurden in der aspektorientierten Variante vier. Die Klassen, in denen diese Aufrufe enthalten waren, konnte teilweise komplett gelöscht werden.

Alle bei der Analyse des Status quo genannten Probleme können mit der aspektorientierten Variante 2 vermieden werden. Das heißt, der Erweiterbarkeit durch Entwickler und der Benutzung aller Ebenen durch Anwendungsprogramme steht jetzt nichts mehr im Wege. Der Synchronisationsaspekt ist durch alternative Aspekte austauschbar und die PURE Subsysteme, auf die diese Aspekte wirken, sind in allen Szenarien wiederverwendbar.

Während der Entwicklung dieser Fallstudien sind darüber hinaus einige Klassen aufgefallen, die kritische Datenstrukturen enthalten, bisher aber keiner Unterbrechungssynchronisation unterlagen. Durch die uniforme Behandlung der Subsysteme in Variante 2 wurden diese Probleme jetzt beseitigt, so dass es zu einer verbesserten Qualität der Software kam.

9.3 Fadensynchronisation

Ein Betriebssystem, das die nebenläufige Abarbeitung von Kontrollfäden erlaubt, muss auch auf die Synchronisation bei der Abarbeitung von Instruktionen durch Fäden achten. In dieser Fallstudie geht es um die Frage, in wie weit Code zur Fadensynchronisation durch Aspekte in ein Betriebssystem eingebracht werden kann und ob die modulare Implementierung der Synchronisationsstrategie wie bei der Unterbrechungssynchronisation auch sinnvoll ist.

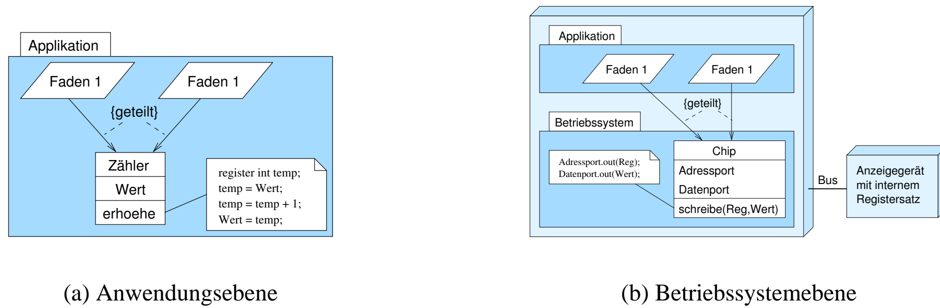
9.3.1 Synchronisationsbedarf

Der Bedarf für Synchronisation entsteht immer dann, wenn zwei oder mehr Fäden ein Betriebsmittel geteilt nutzen. Dies kann innerhalb des Anwendungscodes und auch innerhalb des Betriebssystemcodes geschehen. Auf der Anwendungsebene könnte dies zum Beispiel eine gemeinsam genutzte Zählvariable sein. Auf der Betriebssystemebene benutzen Fäden häufig die gleichen Gerätetreiber, so dass der von den Treibern und den angeschlossenen Geräten verwaltete Zustand ebenfalls hinsichtlich Synchronisation kritisch ist. Abbildung 9.10 auf der nächsten Seite zeigt diese Szenarien in Form von UML Diagrammen.

Die Implementierung der Funktion `erhoehe()` in Teilabbildung (a) mag auf den ersten Blick unnötig aufwändig erscheinen[2]. Sie spiegelt jedoch wider, in welche Teilschritte eine Werterhöhung in der Regel durch den Übersetzer zerlegt wird. Dies ist bei der Betrachtung von Synchronisationsproblemen sehr wichtig. Es kann nämlich nicht davon ausgegangen werden, dass eine Hochsprachenanweisung atomar ausgeführt wird.

Teilabbildung (b) zeigt einen ganz einfachen Gerätetreiber namens `Chip`, der vom Betriebssystem für den Zugriff auf ein angeschlossenes Anzeigegerät bereitgestellt wird. Das angeschlossene Gerät soll in diesem Beispiel über einen internen Registersatz verfügen, der jedoch nicht direkt, sondern nur über einen Adress- und

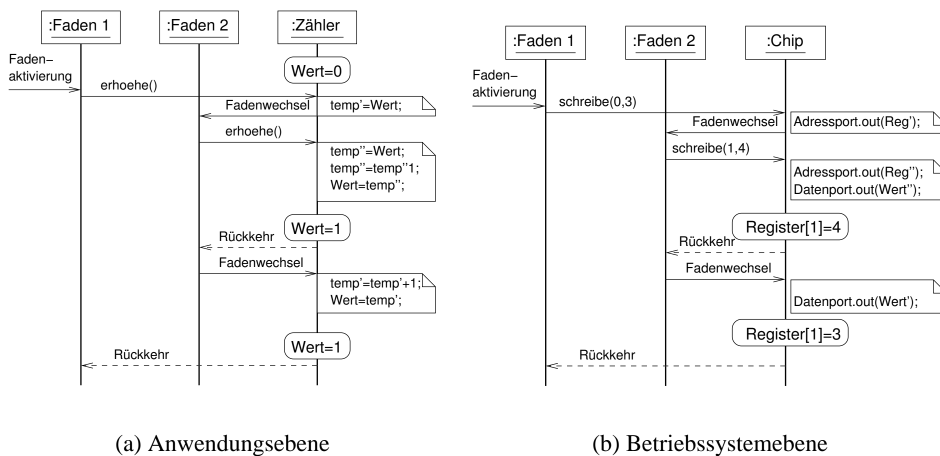
[2]Der geübte C-Programmierer denkt hier sofort an `Wert++`.



(a) Anwendungsebene (b) Betriebssystemebene

Abbildung 9.10: Zugriff auf eine geteilte Betriebsmittel durch Fäden

einen Datenport angesprochen werden kann[3]. Der Treiber stellt keinerlei gerätespezifische Funktionen zur Verfügung, sondern erlaubt lediglich die Manipulation der internen Geräteregister durch die Anwendungsfäden mit Hilfe der Funktion `schreibe(Reg,Wert)`. Dazu muss die Funktion erst die Nummer des Registers in einen Adressport schreiben und danach den Wert für das Register über den Datenport ausgegeben.



(a) Anwendungsebene (b) Betriebssystemebene

Abbildung 9.11: Unerwartetes Verhalten durch fehlende Synchronisation

Abbildung 9.11 zeigt mit Hilfe von Sequenzdiagrammen Szenarien, bei denen durch fehlende Synchronisation ein unerwartetes Verhalten auftritt. Die Teilabbildungen (a) und (b) korrespondieren mit Abbildung 9.10 (a) und (b). In beiden Fällen kommt es zu dem Fehlverhalten, da der Fadenwechsel in einem "ungünstigen" Moment erfolgt. Das Problem tritt also nur auf, wenn Fäden durch externe

[3]Eine solche Architektur kommt gerade im PC Bereich sehr häufig vor, da so der knappe E/A Adressraum nicht unnötigt belegt wird [80].

Ereignisse verdrängt werden können. In diesem Fall kommt es jedoch zu einem nicht-deterministischen Systemverhalten. Wie Teilabbildung (a) zeigt, kann trotz zweimaligen Aufrufs der Funktion `erhoehe()` der Wert nur um 1 erhöht sein. Bei dem Szenario in Teilabbildung (b) überschreibt der Wert, der eigentlich für Register 0 bestimmt war, den Wert, der zuvor in Register 1 eingetragen wurde.

Die Verantwortlichkeit für Synchronisation, die vom Betriebssystem zu übernehmen ist, betrifft natürlich nur den Betriebssystemcode. Für Synchronisation beim Zugriff auf gemeinsam genutzte Datenstrukturen auf Anwendungsebene muss die Anwendung selbst sorgen. Das Betriebssystem kann hier nur Synchronisationsprimitiven zur Verfügung stellen. Damit betrifft die Synchronisationsproblematik in einem objektorientierten System wie PURE alle betriebsmittelverwaltenden Klassen wie beispielsweise Gerätetreiber. Dies schließt natürlich Synchronisationsprobleme mit Zählern wie in Abbildung 9.11 (a) nicht aus, da beispielsweise auch Gerätetreiber Zugriffszähler verwalten könnten.

Bei PURE arbeiten Gerätetreiber bisher unsynchronisiert. Damit dürfen sie jeweils nur von einem Faden exklusiv genutzt werden, was durch die Anwendung sicherzustellen ist.

Um die Möglichkeit einer aspektorientierten Implementierung zu erproben, soll in dieser Fallstudie der Treiber für den PC Disketten-Controller ("*Floppy*-Treiber") und der durch ihn benutzte Treiber für den DMA Chip 8237A um Synchronisationscode erweitert werden. Abbildung 9.12 zeigt dieses Beispielszenario. Während der DMA Treiber nur aus einer Klasse (`DMA8237A`) besteht, wurde der *Floppy*-Treiber auf Basis einer zugrunde liegenden Hierarchie aus Schichten aufgebaut, die, wie bei PURE üblich, durch Klassen und Vererbung implementiert werden. Die letzte Spezialisierung erfolgt durch die Klasse `PCFloppyDriver`, die die Funktionen des Treibers über eine abstrakte allgemeine Blocktreiberschnittstelle zur Verfügung stellt.

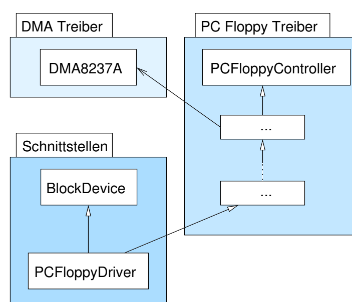


Abbildung 9.12: Beispielszenario "*Floppy*-Treiber"

9.3.2 Gegenseitiger Ausschluss

Der DMA Treiber erlaubt bestimmten Geräten den direkten, von der CPU unabhängigen, Zugriff auf den Hauptspeicher (engl. *direct memory access*). Dazu ist jedoch eine Programmierung des DMA Chips 8237A erforderlich. Es ist durchaus möglich, dass neben dem *Floppy*-Treiber noch ein zweiter Treiber[4] die Klasse `DMA8237A` benutzt, so dass es zu Synchronisationsproblemen kommen könnte, falls die Treiber nebenläufig arbeiten und im DMA Treiber kritische Instruktionsfolgen ausgeführt werden. Dies ist der Fall[5], so dass beim Zugriff auf den DMA Treiber ein gegenseitiger Aussschluss (engl. *mutual exclusion*) garantiert werden muss.

Eine bewährte Möglichkeit für gegenseitigen Ausschluss zu sorgen, besteht in der Anwendung eines Semaphors, der, wie im folgenden Codefragment gezeigt wird, den kritischen Abschnitt schützt:

```
Semaphore mutex(1); // Initialisierung mit Zählwert 1

//...
mutex.wait();   // Auf das Freiwerden warten und betreten
<kritischer Abschnitt>
mutex.signal(); // Kritischen Abschnitt freigeben
```

Mit Hilfe eines einfachen Aspekts ließe sich ein solcher Schutz "von außen" zum DMA Treiber hinzufügen. Die Idee ist dabei, dass die Klasse `DMA8237A` die zu schützenden Daten kapselt und ihre Methoden die kritischen Abschnitte sind. Abbildung 9.13 auf der nächsten Seite zeigt den entsprechenden Aspektquelltext. Der Aspekt besitzt als Attribut den Semaphor `mutex`, der mit Hilfe des Konstruktors auf den Zählwert 1 initialisiert wird. Der *Pointcut* `funcs()` beschreibt alle Aufrufe von Methoden der Klasse `DMA8237A`, die sich nicht innerhalb der Klasse befinden. Durch den *Advice*-Code für diese Menge von Verbindungspunkten, in dem `mutex.wait()` bzw. `mutex.signal()` aufgerufen wird, wird sichergestellt, dass sich jeweils nur ein Faden innerhalb der Klasse `DMA8237A` befinden kann. Die Klasse `DMA8237A` bildet unter Einwirkung des Aspekts einen Monitor [55] ohne Ereignisvariablen, ohne dass der Quellcode dafür modifiziert werden müsste.

Auch innerhalb des *Floppy*-Treibers könnte Nebenläufigkeit fatale Auswirkungen haben. Wenn zum Beispiel ein Faden bei der Programmierung der Geräteregister des Disketten-Controllers unterbrochen wird und ein anderer Faden aktiviert wird, der diese Register ebenfalls programmiert, würden die zuerst eingetragenen Werte unweigerlich verloren gehen. Nach der Rückkehr zum ersten Faden

[4]Bei heutigen PC Architekturen kommt dafür hauptsächlich ein Treiber für eine Soundkarte in Frage.

[5]Zwar stellt der Chip für unterschiedliche DMA Kanäle und damit Treiber unabhängige Register zur Verfügung, doch existiert für den Zugriff auf 16 Bit Register ein globales Bit, das steuert, ob das höher oder das niederwertige Byte geschrieben werden soll. Damit könnte ein Fadenwechsel zu einem ungünstigen Zeitpunkt hier zur falschen Programmierung des Chips und sehr schwer zu lokalisierenden, nicht regelmäßig reproduzierbaren Fehlern führen.

```
aspect DMAMutex {
  Semaphore mutex;
  pointcut funcs() = call("DMA8237A") && !within("DMA8237A");
  DMAMutex() : mutex(1) {} // Semaphor Initialisierung
  // Abfangen und Schutz aller Aufrufe
  advice funcs() : before () { mutex.wait (); }
  advice funcs() : after ()  { mutex.signal (); }
};
```

Abbildung 9.13: Aspekt zum gegenseitigen Aussschluss beim Zugriff auf den DMA Treiber

käme es zu einem Fehlverhalten[6]. Um auch hier für gegenseitigen Ausschluss zu sorgen, könnte der Aspekt `DMAMutex` zu einem gemeinsamen Synchronisationsaspekt `FloppyAndDMAMutex` ausgebaut werden. Dazu müsste lediglich der Pointcut `funcs()` geändert werden, wie es in Abbildung 9.14 zu sehen ist.

```
pointcut drivers() = "DMA8237A" ||
  (derived("PCFloppyController") && base("PCFloppyDriver"));
pointcut funcs() = call(drivers()) && !within(drivers());
```

Abbildung 9.14: Erweiterung des Synchronisationsaspekts unter Beachtung des *Floppy*-Treibers

Das Ergebnis dieser Änderung wäre der Schutz des *Floppy*-Treibers und des DMA Treibers über einen gemeinsamen Semaphor. Hierbei ist jedoch zu kritisieren, dass die Nebenläufigkeit mehr als notwendig eingeschränkt wird. So müsste ohne zwingenden Grund ein Faden warten, der aus einem anderen Treiber kommt und im Begriff ist, den DMA Chip zu programmieren, nur weil zur Zeit ein anderer Faden den *Floppy*-Treiber durchläuft und dort vielleicht die Register des *Floppy*-Controllers programmiert. Ähnlich ist die Situation, wenn in einem System mehrere *Floppy*-Controller existieren und daher mehrere *Floppy*-Treiber Instanzen gebraucht werden. Da ein Aspekt in AspectC++ standardmäßig nur einfach instanziiert wird, teilen sich alle Treiberinstanzen denselben Semaphor. Die *Floppy*-Controller könnten damit nie gleichzeitig benutzt werden. Um dieses Problem zu vermeiden, zeigt Abbildung 9.15 auf der nächsten Seite eine zweite Lösung. Dabei wurde der eigentliche Synchronisationsmechanismus in einen wiederverwendbaren Basisaspekt ausgelagert. In einem abgeleiteten konkreten Aspekt wird nun

[6]Ein solches Szenario ist bei einem *Floppy*-Treiber zwar eher unüblich aber dennoch denkbar. So könnte neben dem normalen Datentranfer mit Hilfe einer Dateisystemimplementierung noch ein weiterer Faden existieren, der in regelmäßigen Abständen den sogenannten *Boot*-Block liest und auf Virenbefall untersucht.

festgelegt, wo Aspektinstanzen und damit Semaphore plaziert werden sollen. Dasselbe Implementierungsmuster wurde bereits in Abschnitt 7.3.3 einmal vorgestellt. Dabei sorgt eine Einfügung für die Entstehung der Aspektinstanzen und die statische Methode `aspectOf()` definiert, wie vom jeweiligen Verbindungspunkt aus die passende Aspektinstanz zu finden ist. In Abbildung 9.15 erfolgt die Einfügung über den *Pointcut* `inst()` in die Klassen `DMA8237A` und die *Floppy*-Treiber Basisklasse `PCFloppyController`. Jedes Treiberobjekt enthält dadurch eine Aspektinstanz und somit ein Semaphor-Objekt. Durch diese Implementierung wird eine Trennung der Definition der zu schützenden Regionen vom Schutzmechanismus erreicht. Beides könnte damit nun unabhängig konfiguriert werden.

```
aspect Mutex {
  Semaphore mutex;
  pointcut virtual funcs() = 0;
  Mutex() : mutex(1) {} // Semaphor Initialisierung
  // Abfangen und Schutz aller Aufrufe
  advice funcs() : before () { mutex.wait (); }
  advice funcs() : after ()  { mutex.signal (); }
};

aspect FloppyAndDMAMutex : public Mutex {
  pointcut inst()   = "DMA8237A"||"PCFloppyController";
  pointcut floppy() = derived("PCFloppyController") &&
                      base("PCFloppyDriver");
  pointcut funcs()  =
    (call("DMA8237A") && !within("DMA8237A")) ||
    (call(floppy()) && !within(floppy())
  advice inst(): FloppyAndDMAMutex __inst;
  static FloppyAndDMAMutex *aspectOf () {
    return &thisJoinPoint->target()->__inst;
  }
};
```

Abbildung 9.15: Aspekt zum gegenseitigen Aussschluss beim Zugriff auf den DMA- und den *Floppy*-Treiber

9.3.3 Leser-/Schreiber-Synchronisation

Der gegenseitige Ausschluss ist eine sehr einfache Form der Synchronisation. In der Praxis braucht man noch flexiblere Synchronisationsmöglichkeiten. Beispielsweise hat der *Floppy*-Treiber einige Methoden, die nur bestimmte Treiberattribute abfragen, und sich gegenseitig bei nebenläufiger Bearbeitung nicht stören würden. Beim gegenseitigen Ausschluss wird diese Nebenläufigkeit unterbunden. Dieses Szenario ist als Leser/Schreiber-Problem bekannt geworden [28] und in der

Literatur finden sich dementsprechend zahlreiche Lösungsmöglichkeiten für die Synchronisation von lesenden und schreibenden Fäden [100, 98]. Abbildung 9.16 zeigt eine aspektorientierte Implementierung auf Basis einer Lösung des ersten Leser/Schreiber-Problems[7].

```
aspect ReadersAndWriters {
  pointcut virtual readers() = 0;
  pointcut virtual writers() = 0;
  Semaphore x, wsem;
  int readcount;
  ReadersAndWriters() : x(1), wsem(1), readcount(0) {}
  // Synchronisationsverhalten der Leser
  void before_readers() {
    x.wait();
    readcount++;
    if (readcount==1) wsem.wait();
    x.signal();
  }
  void after_readers() {
    x.wait();
    readcount--;
    if (readcount==0) wsem.signal();
    x.signal();
  }
  // Advice-Code
  advice readers() : before() { before_readers(); }
  advice readers() : after()  { after_readers(); }
  advice writers() : before() { wsem.wait(); }
  advice writers() : after () { wsem.signal(); }
};
```

Abbildung 9.16: Aspekt zur Leser/Schreiber-Synchronisation entsprechend dem ersten Leser/Schreiber-Problem

Der in der Abbildung 9.16 gezeigte Aspekt benutzt rein-virtuelle *Pointcuts*, kann also nicht instanziiert werden, ohne dass noch ein konkreter Aspekt davon abgeleitet wird. Dies könnte ähnlich wie in Abbildung 9.15 geschehen. Der einzige Unterschied ist, dass hier statt des einen *Pointcuts* `func()` nun die *Pointcuts* `readers()` und `writers()` unterschieden werden müssen.

An diesem Beispiel ist zu erkennen, dass Synchronisationscode zwar durch

[7]Beim *ersten* Leser/Schreiber-Problem dürfen Leser den kritischen Abschnitt immer betreten, wenn mindestens ein anderer Leser gerade darin ist. Dies gilt auch, wenn ein Schreiber bereits wartet. Beim *zweiten* Leser/Schreiber Problem würde der Schreiber dann vorgezogen werden. Im ersten Fall kann es zur Aushungerung (engl. *starvation*) der Schreiber und im zweiten zur Aushungerung von Lesern kommen. Beide Varianten sind gleichermaßen leicht mit einem Aspekt zu implementieren. Wegen der Kürze wird hier die erste gezeigt.

Aspekte von außen in den Komponentencode eingebracht werden kann, jedoch muss dazu Wissen über den Synchronisationsbedarf einzelner Funktionen vorhanden sein. Weitere Erörterungen zu dieser Problematik folgen in Abschnitt 9.3.6.

9.3.4 Nachrichtenbasierte Kommunikation zur Synchronisation

Auch mit Hilfe von blockierender Kommunikation über Nachrichten können Fäden synchronisiert werden [105]. Dabei geht man üblicherweise von zwei Kommunikationsprimitiven aus:

send(Empfänger,&Nachricht) sendet eine Nachricht "Nachricht" an den Faden "Empfänger".

Sender=receive(gewünschter_Sender,&Nachricht) wartet auf das Eintreffen einer Nachricht vom Faden "gewünschter_Sender". Statt auf einen bestimmten Sender kann auch auf Nachrichten mit beliebigem Absender gewartet werden. Dann ist als gewünschter Sender "ANY" anzugeben. Der tatsächliche Sender einer erhaltenen Nachricht wird als Rückgabewert geliefert.

Der Einfachheit halber soll hier davon ausgegangen werden, dass der Parameter "Nachricht" auch weggelassen werden kann, wenn eine leere Nachricht versendet werden soll bzw. sich ein Empfänger für die an ihn geschickte Nachricht inhaltlich nicht interessiert.

Mit Hilfe dieser Primitiven ist es zum Beispiel möglich, das Verhalten des gegenseitigen Ausschlusses eines Monitors durch einen *Server*-Faden nachzubilden. Dieser befindet sich in einer Schleife, in der er Nachrichten mit Arbeitsaufträgen entgegennimmt, diese bearbeitet und dann wieder auf neue Aufträge wartet. Durch dieses Verhalten ist sichergestellt, dass zu jedem Zeitpunkt immer nur ein Auftrag bearbeitet wird. Statt eine per Aspekt synchronisierte Methode aufzurufen, müssten in diesem Fall allerdings die Klienten alle nötigen Parameter für die Funktion und einen Funktionscode in eine Nachricht verpacken und an den *Server*-Faden schicken. Danach müsste der Klienten-Faden auf eine Antwortnachricht warten, die vom Server geschickt wird, wenn die Aufgabe bearbeitet wurde. Damit würde das gleiche Synchronisationsverhalten wie mit der semaphorbasierten `Mutex` Implementierung in Abbildung 9.15 erzielt werden.

Abbildung 9.17 auf der nächsten Seite zeigt eine nachrichtenbasierte Variante des `Mutex` Aspekts und die für seine Implementierung benötigte Hilfsklasse `Server`. Sie erbt von der PURE Klasse `Worker`, wodurch Instanzen von `Server` automatisch aktive Objekte werden. Das sind Objekte mit einem eigenen Kontrollfaden. Die Methode `action()` ist der Startpunkt für den Faden. Er führt die bereits beschriebene Schleife aus, in der fortwährend Arbeitsaufträge empfangen, bearbeitet und bestätigt werden.

Wichtig ist, dass die Klasse `Server` und der Aspekt `Mutex` völlig unabhängig davon sind, welche Methoden im Rahmen der Bearbeitung von Aufträgen ausgeführt werden. Die hierfür nötige Unterscheidung übernimmt die Methode `trigger()` der

```
class Server : public Worker {
  Server() : Worker(STACK_SIZE) { alive(); }
  ~Server() { kill(); }
  void action() {
    while(true) {
      AC::Action *action;
      Worker &sender=receive(ANY,action);
      action->trigger(); // Auftrag bearbeiten
      send(sender);      // Fertigstellung signalisieren
} } };
aspect Mutex {
  pointcut virtual funcs() = 0;
  Server server;
  void synch(AC::Action &action) {
    send(server, &action); // sende Arbeitsauftrag
    receive(server);       // erwarte Fertigstellung
  }
  advice funcs() : around() {
    synch(thisJoinPoint->action());
} };
```

Abbildung 9.17: Aspekt für den gegenseitigen Ausschluss mit Hilfe eines *Server*-Fadens

Klasse `AC::Action`, die von AspectC++ beigesteuert wird. Ein *Action*-Objekt kapselt, wie in Abschnitt 7.3.4 beschrieben wurde, alle nötigen Informationen, um eine durch einen Aspekt unterbrochene Aktion, etwa das Aufrufen einer Funktion, innerhalb des *Advice*-Codes fortzusetzen. Die Erzeugung eines solchen Objekts erfolgt auf Anforderung, wenn `thisJoinPoint->action()` aufgerufen wird. Ein *Action*-Objekt enthält unter anderem Zeiger auf alle Parameter eines unterbrochenen Funktionsaufrufs, so dass das Zusammenbauen einer entsprechenden Nachricht durch den `Mutex` Aspekt hier nicht nötig ist.

Durch eine Konfigurierungsmaßnahme, bei der wahlweise die semaphorbasierte oder die nachrichtenbasierte `Mutex` Variante gewählt werden kann, können der DMA- oder der *Floppy*-Treiber nun auch durch einen *Server*-Faden bearbeitet werden.

Neben dem `Mutex` Aspekt wurde im Rahmen dieser Fallstudie auch eine nachrichtenbasierte Variante des `ReadersAndWriters` Aspekts implementiert. Dabei wurde der Server vielfädig ausgelegt, wobei ein Faden für die Annahme der Aufträge und die Ausführung von schreibenden Operationen verantwortlich war, während die restlichen Fäden einen Pool zur Ausführung von Leseaufträgen bildeten. Lesende und schreibende Fäden mussten dann wiederum synchronisiert werden, wobei die semaphorbasierte Leser/Schreiber-Synchronisation zum Einsatz kam. Auf weitergehende Ausführungen zu dieser Synchronisationsvariante wird verzichtet, da

sie für das weitere Verständnis nicht nötig sind.

9.3.5 Vergleich der Varianten

		keine Synch.	Server-Faden		Semaphor	
			RW/Mutex	RW/–	RW/Mutex	RW/–
Zeiten (Zyklen)	seek	18650	19835	19828	18924	18810
	transfer	65463	66823	65006	65196	64764
	finish	20518	21194	22042	20771	20642
	dma	2914	3948	2856	3109	2940
Größe (Bytes)	text	14869	19898	18742	15901	15733
	data	20	32	32	20	20
	bss	1072	1088	1080	1092	1080
	total	15961	21018	19854	17013	16833

Tabelle 9.2: Vergleich des Ressourcenverbrauchs der Synchronisationsvarianten

Tabelle 9.2 zeigt einen Vergleich des Laufzeitverhaltens[8] und der Codegrößen verschiedener Testprogramme, die mit Hilfe der in den vorangegangenen Abschnitten vorgestellten Aspekte implementiert wurden. "RW/Mutex" bedeutet eine Synchronisationsvariante, bei der der *Floppy*-Treiber mit Leser/Schreiber-Synchronisation arbeitet und der DMA-Treiber mit gegenseitigem Ausschluss. Bei "RW/–" erfolgt beim Zugriff auf den DMA-Treiber keine Synchronisation. Nach einigen technisch notwendigen Schritten zur Systeminitialisierung bestand die Aufgabe der Testprogramme jeweils darin, einen Block von einer Diskette einzulesen. Abbildung 9.18 auf der nächsten Seite erläutert die Bedeutung der Phasen "seek", "transfer", "dma" und "finish", deren jeweilige Dauer in der Tabelle dargestellt ist. Für das Lesen ein Blocks von der Diskette kommt es innerhalb des *Floppy*-Treibers zweimal zur Blockierung des ausführenden Fadens, da auf die langsame Mechanik des Laufwerks gewartet werden muss. Diese in jeder Variante gleichen Wartezeiten werden durch die gemessenen Zeitparameter nicht erfasst, da hier vor allem der Verbrauch an CPU-Zeit interessiert.

Wie in der Tabelle zu erkennen ist, machen sich die unterschiedlichen Synchronisationsvarianten vor allem auf die Dauer des Zugriffs auf den DMA-Treiber bemerkbar (Parameter "dma"), da diese Phase relativ kurz ist. Die Unterschiede bei den anderen Parametern liegen nur bei wenigen Prozent. Absolut gesehen lässt sich herauslesen, dass bei der Interaktion mit einem *Server*-Faden eine zeitliche Belastung in einer Größenordnung von etwa 1000 Takten entsteht, während die Semaphore-basierte Synchronisation nur mit etwa 200 Takten zu Buche schlägt.

Eine deutlichere Sprache sprechen die Zahlen zum Speicherplatzverbrauch. Hier verbrauchen die Varianten mit Server-Faden einige KBytes mehr, wobei der

[8] Pentium III (300MHz) Taktzyklen

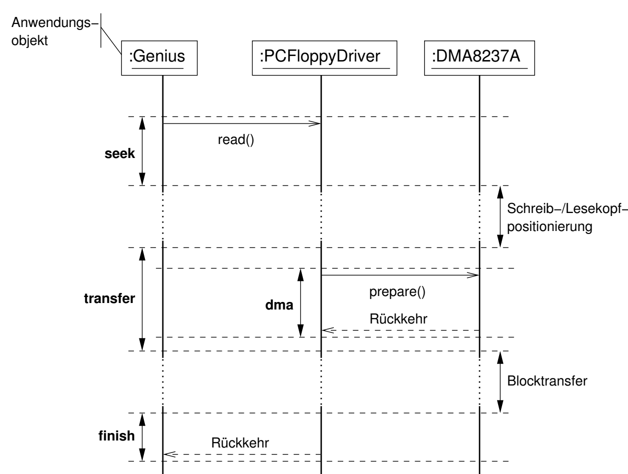


Abbildung 9.18: Phasen und Objektinteraktion beim Lesen eines Diskettenblocks

dynamisch angeforderte Speicherplatz für die Stapelspeicher der Fäden noch nicht einmal eingerechnet ist.

Obwohl die Varianten mit *Server*-Fäden offensichtlich einen größeren Ressourcenverbrauch aufweisen, kann ihr Einsatz in vielen Fällen durchaus sinnvoll sein. So wäre es möglich, statt eines *Server*-Fadens einen *Server*-Prozess mit eigenem Adressraum zu benutzen oder einen Faden, der die ausschließlichen Zugriffsrechte auf die mit dem Gerät verbundenen Register besitzt. Damit kann ein Schutz der Hardware-Betriebsmittel und des Treibers vor unerlaubtem oder ungewolltem Zugriff erfolgen und andererseits könnte das System vor Fehlern im Treiber geschützt werden, falls dieser in einem isolierten Adressraum arbeitet. Das System wird durch solche Schutzmaßnahmen robuster. Dieser und weitere Punkte sind typische Argumente für die sogenannte Mikrokernarchitektur von Betriebssystemen, bei denen auch Gerätetreiber als eigene Prozesse oder Fäden oberhalb eines kleinen Systemkerns arbeiten [18]. Diese Gemeinsamkeit des hier erzielten Ergebnisses mit dem Mikrokernansatz wird in Kapitel 10 nochmals aufgegriffen und vertieft.

9.3.6 Fadensynchronisationsstrategie

Das Ziel der aspektorientierten Realisierung des Synchronisationsverhaltens von Treibern besteht darin, den kompletten Synchronisationscode möglichst kompakt und modular auszudrücken, so dass die damit verbundene Synchronisationsstrategie sichtbar wird. Mit den hier vorgestellten Mitteln ist dies nun möglich. So können die in jeweils zwei Varianten vorliegenden Aspekte `Mutex` und `ReadersAndWriters` zu einer ganzen Bibliothek von abstrakten Synchronisationsaspekten ausgebaut werden. Konkrete Synchronisationsaspekte bestimmten durch das Erben von einem abstrakten Aspekt aus dieser Bibliothek die Synchronisa-

tionsimplementierung und über die Erzeugung von Aspektinstanzen die Zuordnung zu Regionen, deren Bearbeitung synchronisiert werden muss. Dies wurde in Abbildung 9.15 schon gezeigt. Dort wurde allerdings mit `FloppyAndDMAMutex` ein spezieller Aspekt für genau zwei Treiber geschaffen. Zur Implementierung einer Strategie würde man weiter gehen und möglichst alle Treiber aufführen, die mit gegenseitigem Ausschluss synchronisiert werden sollen. In gleicher Weise würden alle Treiber mit Leser/Schreiber-Synchronisation oder anderen Synchronisationsvarianten aufgeführt werden. Dabei muss die Zuordnung von einem Treiber zu einer Synchronisationsvariante nicht fest sein. Mittels *Pointcuts* können die Funktionen eines Treibers bezüglich ihrer Synchronisationsanforderungen kategorisiert werden. Wie Abbildung 9.19 auf der nächsten Seite im Einzelnen zeigt, können so die Treiber frei den Synchronisationsaspekten wie `Mutex` oder `ReadersAndWriters` zugeordnet werden.

Durch die kompakte und modulare Implementierung ist die Synchronisationsstrategie auf diese Weise überschaubar und konfigurierbar, so dass anwendungsspezifische Optimierungen gemacht werden können. Wenn beispielsweise feststeht, dass der DMA Treiber nicht von der Anwendung und intern nur vom *Floppy*-Treiber benutzt wird, kann dort auf Synchronisation verzichtet werden, falls beim *Floppy*-Treiber bereits für gegenseitigen Ausschluss gesorgt wird. Darüber hinaus ist es so sehr leicht möglich, Klassen ohne Synchronisationscode für eine vielfädige Ausführungsumgebung zu erweitern. Aus einer einfachen Listenklasse wird so schnell unter Zuhilfename eines *Producer/Consumer*-Aspekts eine *Mailbox* zur synchronisierten indirekten Interprozesskommunikation.

9.3.7 Bewertung

Wie die letzten Abschnitte gezeigt haben, ist es mit Aspekten möglich, durch *Advice*-Code für Synchronisation zu sorgen, wenn Übergänge zwischen Klassen stattfinden. Unter Umständen ist auch eine feinere Kontrolle möglich. Dies führt jedoch wieder zu dem Problem, dass Aspekte Wissen über die Implementierung bestimmter Klassen nutzen würden. Damit hängt die Möglichkeit mit Hilfe von Aspektcode zu synchronisieren, in starkem Maße von der Granularität der Funktionen und gegebenenfalls Klassen des Systems ab. Im Falle von PURE, wo auf minimale Erweiterungsschritte geachtet wurde, ergänzen sich Synchronisationsaspekte und die bisherigen Systemstrukturen daher sehr gut.

Während der Implementierung der nachrichtenbasierten Varianten trat das Problem auf, dass Treiber zum Teil globale Systemobjekte sind, und dass solche Objekte während ihrer Konstruktion keine Fäden erzeugen dürfen. Dies wäre jedoch für die *Server*-Fäden gewünscht gewesen. Der Grund für das Problem ist, dass die Sprache C++ keine Zusicherungen über die Reihenfolge der Bearbeitung globaler Konstruktoren macht, so dass es sein kann, dass zum Zeitpunkt der Fadenerzeugung die Fadenverwaltung noch nicht initialisiert ist, was in der Regel zu einem Systemabsturz führt. Zur Umgehung des Problems werden die benötigten Fadenobjekte durch die Konstruktoren globaler Treiberobjekte zunächst in eine Liste ein-

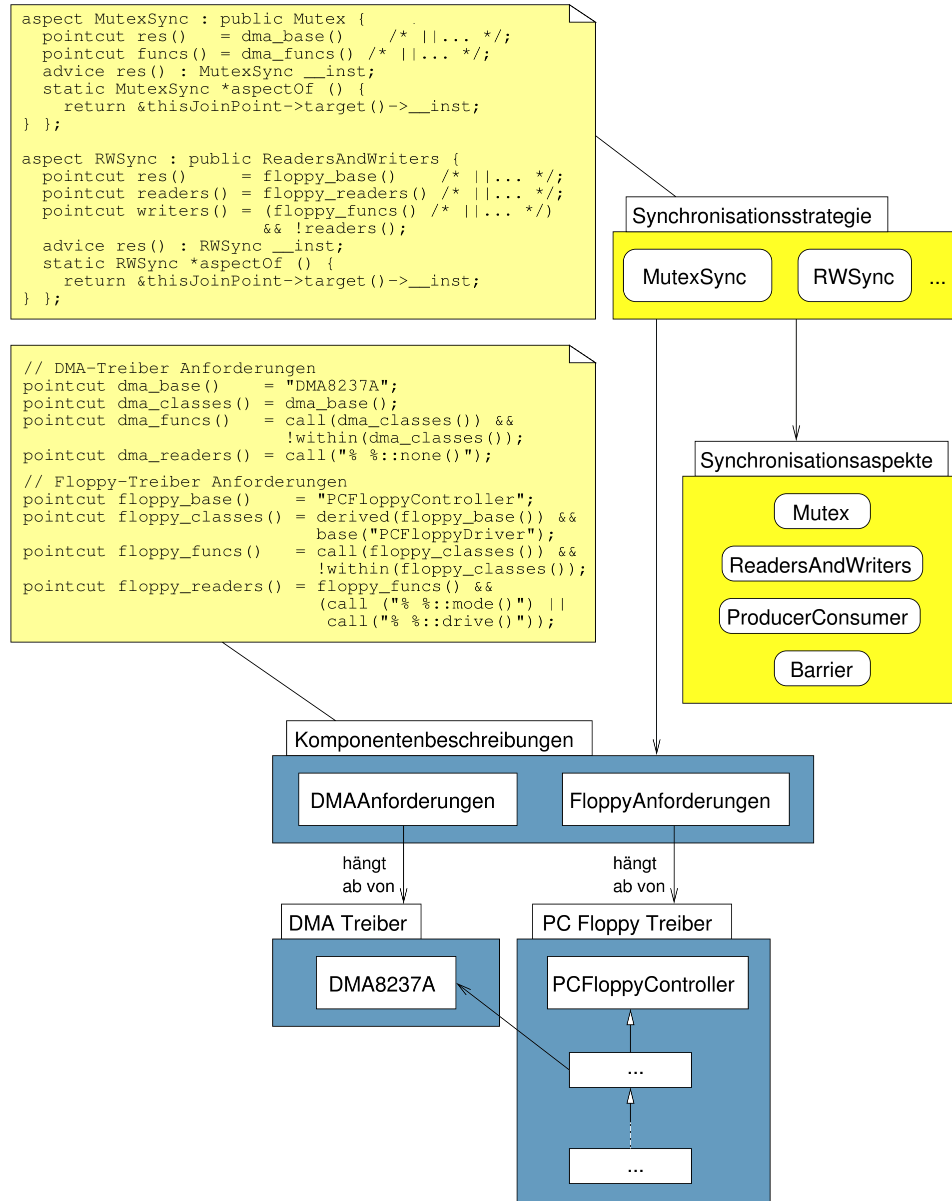


Abbildung 9.19: Strategie zur Fadensynchronisation

getragen, mit deren Hilfe ein zusätzlicher Aspekt die Fäden verzögert startet. Besser als diese komplizierte Lösung wäre die Überarbeitung des kompletten Verhaltens beim Systemstart, so dass weniger oder gar nichts in globalen Konstruktoren erfolgt und stattdessen eine Startphase durchlaufen wird, in der sich die Subsysteme in der richtigen Reihenfolge initialisieren können. Dieses Problem macht erneut deutlich, dass Aspekte und die Komponenten, auf die sie wirken, aus einem Guss sein sollten und dass Aspekte beim Entwurf zu berücksichtigen sind. Bei einem Systementwurf, bei dem davon ausgegangen worden wäre, dass ein konfigurierbares *Crosscutting Concern* "Fadensynchronisation" auf die Treiber des Systems wirkt, hätte man frühzeitig anhand des Feature Modells erkannt, dass es *Server*-Fäden geben kann.

Abgesehen von dieser Erkenntnis zeigt sich bei diesem Beispiel jedoch das Potential des aspektorientierten Ansatzes für die Betriebssystemfamilie. Die normalerweise strategische Entscheidung, wo und wie in einem System Fäden zu synchronisieren sind, fällt hier auf wenige Aspekte zusammen. Diese Fallstudie lässt die Idee einer architekturneutralen Treiberentwicklung realisitisch erscheinen, bei der Treibercode unverändert im Kontext verschiedener konfigurierbarer Systemarchitekturen verwendet werden kann, da die Zuordnung von Fäden, Synchronisationspunkten und später vielleicht auch Adressräumen von außen durch Aspekte erfolgt.

9.4 Zusammenfassung

In diesem Kapitel wurden drei Anwendungsbeispiele im Kontext von PURE und unter Zuhilfenahme von AspectC++ vorgestellt. Im folgenden soll nun auf die Bewertungskriterien für den Ansatz, die in Abschnitt 5.3 genannt wurden, noch einmal kurz Bezug genommen werden. Ausführlichere Erläuterungen der einzelnen Punkte erfolgten bereits in den Bewertungsabschnitten der einzelnen Fallstudien.

1. Eignung der Modellerweiterung: Dieser Frage wurde in dem Analyse- und Entwurfsbeispiel in Abschnitt 9.1 nachgegangen. Dort haben sich die *Concern*-Hierarchien zur Modellierung des "abstrakten Prozesszustands" bewährt. Damit liegt nun ein brauchbarer Grundbaustein für weitere methodische Arbeiten an aspektorientierten Programmfamilien vor.

2. Funktionstüchtigkeit der Werkzeuge: Im Rahmen der zweiten und dritten Fallstudie wurden AspectC++ und SOSP eingesetzt. Darüber hinaus wird AspectC++ für die Übersetzung von PUMA verwendet. Man kann also von der Funktionstüchtigkeit ausgehen.

3. Modulare Implementierung globaler Strategien: Dieser Punkt wurde durch die Fallstudien zur Unterbrechungs- und Fadensynchronisation gezeigt. Damit stellt sich nun die Frage nach weiteren globalen Strategien, die aspektorientiert behandelt werden können. In Abschnitt 10.3 wird dazu eine Übersicht präsentiert.

4. Reduzierung der sichtbaren Konfigurationspunkte: Diese These kann bloß über Vergleichsstudien belegt werden. Dafür kommt nur die Unterbrechungssynchronisationsstudie in Abschnitt 9.2 in Frage. In der Status quo Variante war die Unterbrechungssynchronisation jedoch nicht konfigurierbar. Bedenkt man aber, dass bisher an 166 Punkten in 15 Klassen Synchronisationsprimitiven aufgerufen wurden, so ist sicher von einer drastischen Reduzierung der Zahl der Konfigurationspunkte im Vergleich zu einer herkömmlichen Implementierung mit der gleichen Variabilität auszugehen.

5. Reduzierung des nötigen Quellcodes: In der Fallstudie zur Unterbrechungssynchronisation kam es durch die Eliminierung der vielen Synchronisationsaufrufe, einiger Klassen und Klassenaliase zu einer Reduzierung des Quellcodes.

6. Realisierung ohne zusätzlichen Ressourcenverbrauch: Aufgrund der nur auf Anforderung erfolgenden Codegenerierung durch AspectC++ kommt es im Allgemeinen nicht zu einem erhöhten Ressourcenverbrauch durch Aspekte. Dies konnte mit Größenmessungen in Abschnitt 9.2 bestätigt werden.

Besonders wichtig scheint in dieser Liste der dritte Punkt zu sein. Betriebssysteme, bei denen abstrakte Prozesszustände, die Unterbrechungssynchronisation oder die Fadensynchronisation in ähnlicher Weise konfigurierbar wären, wie es hier demonstriert wurde, ohne dass entweder die Verständlichkeit der Systemkomponenten oder der Ressourcenverbrauch darunter drastisch leiden, existieren bisher nicht.

Kapitel 10

Ausblick

Die in den vorangegangenen Kapiteln 6 bis 8 vorgestellten Werkzeuge eröffnen ein breites Spektrum an Folgearbeiten. Zudem haben die Fallstudien neben der Bestätigung, dass der in dieser Arbeit verfolgte Ansatz in die Tat umgesetzt werden kann, einige Probleme aufgezeigt, an denen in Zukunft gearbeitet werden sollte. Dieses Kapitel gibt daher einen ausführlichen Ausblick auf zukünftige Arbeiten und einige Antworten auf die erkannten Probleme.

10.1 Änderungen und Erweiterungen an AspectC++

Zahlreiche Änderungen und Erweiterungen an AspectC++ wären sinnvoll. Davon wurden bereits einige in Abschnitt 7.3.5 genannt und andere wird es mit Sicherheit geben, um die Menge der abdeckbaren *Crosscutting Concerns* zu erhöhen und den Umgang damit zu vereinfachen.

Durch den Einsatz im Bereich eingebetteter Systeme und die zunehmende Verbreitung von AspectC++ durch den frei verfügbaren Übersetzer besteht in Zukunft eine sehr wichtige Frage darin, ob AspectC++ auch für die Entwicklung sicherheitskritischer Software [72] eingesetzt werden kann oder sollte. Schließlich befindet sich eine große Zahl eingebetteter Systeme in sicherheitskritischen Produkten wie Automobilen, Flugzeugen oder Kraftwerken. Daher befassen sich die nächsten Abschnitte zunächst mit der Spezifikation und Verifikation von AspectC++ Programmen sowie notwendigen Spracherweiterungen, um die Risiken, die mit den Vorteilen aspektorientierter Programmierung einhergehen, zu minimieren.

10.1.1 Spezifikation und Verifikation

Aspekte sind ein sehr mächtiges sprachliches Instrument, da sie erlauben, quasi von außen in den Kontrollfluss und die Struktur von Komponentencode einzugreifen, ohne dass dieser speziell dafür prepariert sein müsste. Dabei ist es sogar möglich, das Verhalten der Komponenten, auf die ein Aspekt wirkt, zu verändern. Oftmals ist dies sogar gewollt. So reagiert beispielsweise die Trigonometriekomponente in

Abbildung 5.4 auf Seite 57 anders als sonst auf Argumentwerte über 360 bei `sin()` und `cos()`, wenn der Aspekt `Periode` darauf wirkt. Ein Kontrollflussverfolgungsaspekt wie der in Abbildung 7.4 auf Seite 106 verändert dagegen weder das Ein-/Ausgabe-Verhalten noch die Zustandsänderungen der Komponenten. Es kommt lediglich zu Seiteneffekten in Form der Ausgabe von Protokollinformationen und zu einer zeitlichen Verzögerung.

Wenn im Rahmen der Analysephase eines Entwicklungsprojekts das Verhalten einer Komponente, beispielsweise der Objekte einer Klasse, spezifiziert wird, so muss für eine Verifikation sichergestellt werden, dass auch unter Einwirkung möglicher Aspekte die Komponente in den Grenzen der Spezifikation arbeitet. In Echtzeitsystemen, zu denen kleinste eingebettete Systeme oftmals zählen, gehört zu einer Verhaltensspezifikation neben dem Ein-Ausgabe-Verhalten und dem Zustandsverhalten auch das zeitliche Verhalten einer Komponente [37].

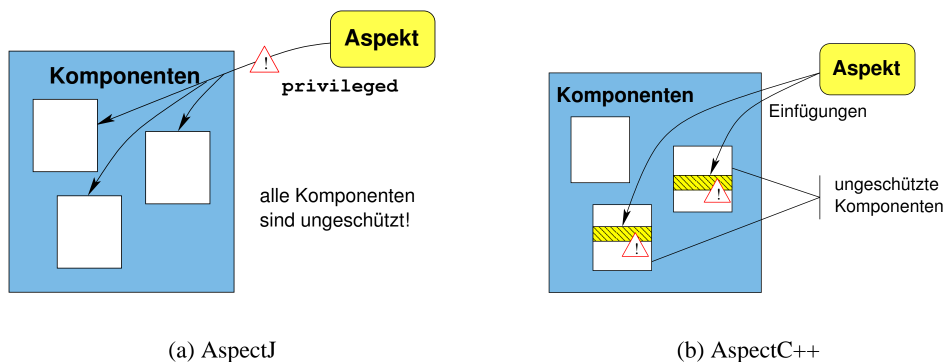
Gerade im Hinblick auf mögliche sicherheitskritische Anwendungen wird intensiv über die Verifizierbarkeit von Software nachgedacht. Damit kann diese Frage in Zukunft auch die Sprache AspectC++ treffen, die für den ressourcensparenden Einsatz im Bereich eingebetteter Systeme konzipiert wurde. Die Verifikation ist ein formaler Nachweis der Korrektheit, das heißt der Erfüllung der Spezifikation, von Programmen oder Programmteilen [69]. Man benötigt dazu die präzise Definition der Semantik des Programms und damit auch der Semantik der Programmiersprache. Um diesen Vorgang zu gestatten, wäre also eine formale Definition der Semantik von AspectC++ notwendig. Dies dürfte sich jedoch als sehr schwierig erweisen, da AspectC++ eine Erweiterung von C++ ist und eine vollständige formale Definition der Semantik von C++ bisher wegen der Komplexität der Sprache und offener Punkte im Standard nicht exisitiert. Gäbe es jedoch eine solche Semantik, dann könnte man mit Hilfe der formalen Definition einer Übersetzersemantik von AspectC++ nach C++ die vollständige Sprache abdecken. Bisher existiert eine solche Definition noch nicht bzw. nur in Form der Implementierung des AspectC++ Übersetzers, da die ausgeführten Transformationen noch gelegentlich Änderungen zum Opfer fallen. Für die Zukunft wäre eine formale Definition der Übersetzersemantik jedoch hilfreich, schon um Anwendern der Sprache deren Bedeutung präzise vermitteln zu können.

Eine weiterer interessanter Ansatz bestünde darin, auf den Sprachdialekt *Safe* C++ zu setzen. Dabei handelt es sich um eine Teilmenge von C++, deren Semantik formal definiert wird [57]. Der AspectC++ Übersetzer könnte dahingend erweitert werden, dass optional als Komponentensprache nur noch *Safe* C++ zugelassen wird.

10.1.2 Schutzkonzept

Eine wesentliche Sprachunterstützung für die Entwicklung sicherer Softwaresysteme betrifft das Schutzkonzept zwischen Aspekten und Komponentencode. Dieser Punkt wurde im Abschnitt 7.3.3 bereits angesprochen. Dort wurde insbesondere der Unterschied zwischen dem Schutzkonzept von AspectJ und AspectC++ erläu-

tert. Abbildung 10.1 verdeutlicht diesen Unterschied nochmals. Bei AspectJ haben Aspekte standardmäßig keine besonderen Zugriffsrechte auf die Attribute oder Methoden von Klassen. Wie Teilabbildung (a) zeigt, hat ein Aspekt jedoch das volle Zugriffsrecht auf alle Klassen, sofern er mit dem Schlüsselwort `privileged` zu einem privilegierten Aspekt ernannt wurde. Bei AspectC++ hat ein Aspekt zunächst ebenfalls keine speziellen Rechte. Diese können aber, wie Teilabbildung (b) zeigt, indirekt durch Einfügungen erlangt werden, da eingefügte Methoden die gleichen Zugriffsrechte besitzen wie herkömmliche Methoden.



(a) AspectJ (b) AspectC++

Abbildung 10.1: Schutzkonzepte von AspectJ und AspectC++ im Vergleich

Für die Verifikation einer Komponente sind Aspekte, die auf deren Interna zugreifen können, kritisch, da sie das Verhalten der Komponente beeinflussen können. Es wäre daher hilfreich, wenn Aussagen darüber, ob ein Aspekt besondere Zugriffsrechte auf eine Komponente besitzt, leicht zu machen wäre, so dass möglichst viele Aspekte von vornherein außer Acht gelassen werden können. Bei dem Schutzkonzept von AspectJ kommt man ohne aufwändige Analysen nur zu der Aussage, dass alle Komponenten von allen privilegierten Aspekten beeinflusst werden können. Bei AspectC++ ergeben sich präzisere Aussagen, doch müssen dazu die *Pointcuts* aller Einfügungen ausgewertet werden. Außerdem fehlt eine explizite Deklaration der Aspektprivilegien, so dass Einfügungen nicht hinsichtlich einer vorher festgelegten Schutzpolitik überprüft werden können. Somit kann es durch Nachlässigkeit beim Programmieren zu einem schrittweisen Aufweichen des Schutzes kommen. Beide Varianten sind also nicht sehr befriedigend.

Ziel zukünftiger Arbeiten an der Sprache AspectC++ sollte daher die Entwicklung eines verbesserten Schutzkonzepts sein. Bei diesem sollte eine Schutzpolitik erklärt werden können, so dass alle Zugriffe innerhalb von *Advice*-Code und Einfügungen auf Konformität getestet werden können. Möglicherweise würde dazu ein um einen *Pointcut* erweitertes Schlüsselwort `privileged` bereits ausreichen. Dabei würde der *Pointcut* beschreiben, welche Klassen, Attribute und Methoden vom jeweiligen Aspekt in ihrem Verhalten verändert werden dürfen. Wichtig ist dabei auch, dass sichergestellt wird, dass ein Aspekt, der gegenüber einer bestimmten

Klasse nicht privilegiert ist, weder ihren internen Zustand noch ihr Ein-/Ausgabe-Verhalten ändern kann. Letzteres ist durch *Advice*-Code und die dazugehörige Laufzeitunterstützung bisher in jedem Fall möglich, indem Parameterwerte oder Rückgabewerte manipuliert werden. In Zukunft sollten diese Objekte im Fall eines nicht-privilegierten Einwirkens konstant sein.

Mit einem solchen Schutzkonzept müssten viele Aspekte bei der Verifikation von Komponenten leicht abzuhandeln sein, da sie keine Möglichkeit hätten, Verhaltensänderungen zu bewirken. Alle anderen Aspekte müssten allerdings unter Verwendung der Übersetzersemantik berücksichtigt werden — im Kontext einer Programmfamilie sogar in allen denkbaren Kombinationen. Gleiches gilt nicht nur für die Verifikation, sondern auch für den Test. Im Kontext von Echtzeitsystemen sind bei der Verifikation von zeitlichen Spezifikationen von Komponenten alle einwirkenden Aspekte zu berücksichtigen, auch wenn diese auf das Ein-/Ausgabe-Verhalten und Zustandsänderungen keinen Einfluss haben.

10.1.3 Erkennung von Konfigurierungsfehlern

Eine weitere wichtige geplante Spracherweiterung, die helfen soll, inkorrekte Programme zu vermeiden, besteht in einem Mechanismus zur Erkennung von Konfigurierungsfehlern. Dabei geht es um das Problem der notwendigen Beziehungen von Aspekten, die in Abschnitt 5.1.6 diskutiert und als kritisch eingestuft wurden. Gleichzeitig wurde diese Beziehgungsart aber in der Fallstudie in Abschnitt 9.1 verwendet, so dass davon ausgegangen werden kann, dass solche Beziehungen nicht grundsätzlich verboten werden dürfen. Stattdessen sollte versucht werden, die mit ihnen verbundene Gefahr zu beseitigen. Wenn beispielsweise die Implementierung eines Aspekts davon abhängt, an einem bestimmten Verbindungspunkt zu wirken, könnte es zu einem Fehlverhalten kommen, falls aufgrund einer falschen Systemkonfigurierung oder auch eines einfachen Tippfehlers in der *Pointcut*-Definition der entsprechende Verbindungspunkt nicht existiert. Der Entwickler des Aspekts kennt jedoch die Wichtigkeit dieses Verbindungspunktes für sein Implement. Wenn die Sprache ihm nun die Möglichkeit gäbe, zu deklarieren, dass der *Pointcut*, der diesen Verbindungspunkt beschreibt, nicht leer sein darf, könnte der Übersetzer mit einer entsprechenden Fehlermeldung reagieren. Der zugrundeliegende Konfigurierungsfehler könnte dann gesucht und behoben werden.

10.2 Unterstützung des Entwicklungsprozesses

Damit Entwickler die Konzepte der aspektorientierten Programmierung gewinnbringend einsetzen können, bedarf es einiger Unterstützung, die über die Bereitstellung einer Sprache und eines Übersetzers hinausgeht. So besteht ein großer Bedarf in den Bereichen methodische Unterstützung, Entwurfsmuster und Entwicklungswerkzeuge, die in den folgenden Abschnitten kurz einzeln angesprochen werden.

10.2.1 Methodische Unterstützung

Für die aspektorientierte Entwicklung einer Programmfamilie bedarf es eigentlich einer Entwicklungsmethode, die alle Phasen von der Analyse bis zum Test unterstützt und durchgängig beschreibt. Dabei könnten Feature Modelle, *Concern*-Hierarchien und eine aspektorientierte Programmiersprache eine zentrale Rolle einnehmen. Diese Arbeit ist ein Beitrag zu einer solchen Methode, die jedoch noch weitgehender Ergänzungen bedarf.

10.2.2 Entwurfsmuster

Eine interessante Fragestellung für die Zukunft ist, ob sich anhand der bisher geleisten Entwicklungsarbeiten Entwurfsmuster für die aspektorientierte Programmfamilienentwicklung ablesen lassen. Die Kombination von Entwurfsmustern und Aspektorientierung ist bereits Gegenstand aktueller Forschungsarbeiten [51]. Dabei wird jedoch der Familiengedanke nicht berücksichtigt. Im Folgenden soll daher ein Entwurfsmuster diskutiert werden, das während der Entwicklungsarbeiten zu den im letzten Kapitel vorgestellten Fallstudien aufgefallen ist.

Der Arbeitstitel dieses Musters soll *Firewall* lauten. Es kann angewendet werden, wenn innerhalb des Systems ein Teilbereich identifiziert werden kann, bei dessen Betreten und Verlassen spezielle (gleichartige) Aktionen ausgeführt werden sollen. Die aspektorientierte Implementierung eines solchen *Crosscutting Concerns* nach dem *Firewall* Muster ermöglicht dabei nicht nur die zusammenhängende Definition der auszuführenden Aktionen, sondern vereinfacht auch das Verschieben der Grenzen des betrachteten Teilbereichs.

Mögliche Anwendungsbereiche dieses Entwurfsmusters gibt es gerade im Betriebssystemsektor, zum Beispiel den in der Fallstudie behandelten Schutz kritischer Datenstrukturen vor Unterbrechungen, die Realisierung von Gerätetreibern als eigene Prozesse oder die Interaktion unterschiedlicher Knoten eines verteilten Systems. Aber auch in anderen Domänen treten vergleichbare Fälle auf, zum Beispiel wenn entschieden werden muss, wann bei einer Kommunikation Daten verschlüsselt werden müssen.

Das Konzept dieses sprachlichen *Firewalls* wird in Abbildung 10.2 auf der nächsten Seite illustriert. Die Grundidee besteht darin, zunächst den identifizierten Teilbereich zu beschreiben und dann alle Funktionsaufrufe, die von außen kommen, zu instrumentieren. Für einen erfolgreichen und wartungsarmen Einsatz des *Firewall* Entwurfsmusters sollten allerdings folgende Voraussetzungen erfüllt sein:

- Der zu schützende Bereich muss gut beschrieben werden können, also möglichst ganze Klassen oder Vererbungshierarchien umfassen.

- Die auszuführenden Aktionen müssen einheitlich formuliert werden können, wobei natürlich die Informationen zum jeweiligen *Joinpoint* genutzt werden dürfen.

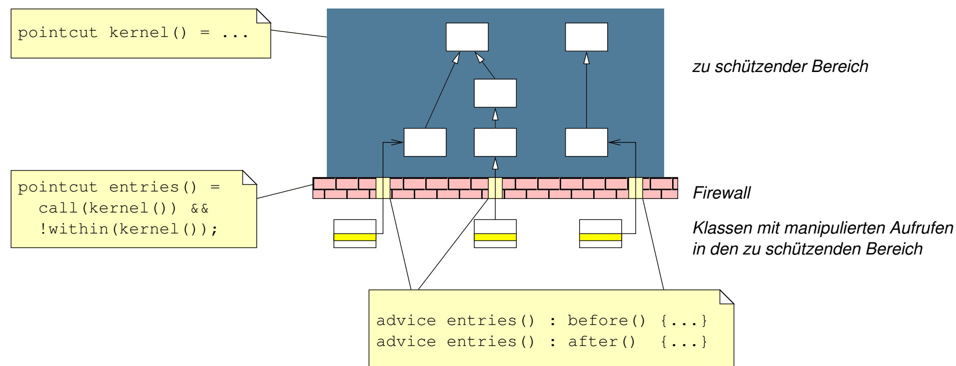


Abbildung 10.2: *Firewall* Entwurfsmuster

Das Beispiel *Firewall* zeigt, dass Muster für den Entwurf aspektorientierter Programmfamilien existieren. Für den ungeübten Entwickler und als Vokabular für den Umgang mit Softwarestrukturen sind solche Muster sehr hilfreich. Daher sollten in der Zukunft noch weitere Muster identifiziert, erprobt und vollständig dokumentiert werden.

10.2.3 Entwicklungswerkzeuge

Entwicklungswerkzeuge können für den Erfolg der aspektorientierten Programmierung eine sehr wichtige Rolle spielen. Beispielsweise können integrierte Entwicklungsumgebungen, die das Kozept der Aspekte kennen, durch eine geeignete Visualisierung anzeigen, auf welche Stellen im Programmcode Aspekte einwirken. Dadurch wird die Gefahr des versehentlichen Löschens wichtiger Verbindungspunkte deutlich verringert.

Die Entwickler von AspectJ haben die Wichtigkeit der Einbettung ihres Übersetzers in integrierte Entwicklungsumgebung schon frühzeitig erkannt, so dass dieser heute mit JBuilder, Forte4J, dem *Editor* Emacs und neuerdings auch Eclipse zusammenarbeitet.

Entsprechende Arbeiten wären natürlich für AspectC++ auch sinnvoll. Daneben wäre es wichtig, die in Abschnitt 10.2.1 angesprochene Entwicklungsmethode durch ein dazu passendes Entwurfswerkzeug (*CASE tool*) zu unterstützen, das neben klassischen UML Diagrammen auch den Entwurf von Feature Modellen und die schrittweise Verfeinerung von *Concern*-Hierarchien gestattet.

10.3 Weitere Anwendungen

Die in dieser Arbeit vorgestellten Fallbeispiele decken nur einen kleinen Teil der *Crosscutting Concerns* in Betriebssystemfamilien ab. Zudem hat sich beispielsweise in der Fallstudie zur Fadensynchronisation gezeigt, dass der Komponentencode möglichst in Hinblick auf das Einwirken von Aspekten entworfen sein sollte (“de-

sign for aspect intervention”). Um die volle Palette der Vorzüge der aspektorientierten Programmierung für den Bau von Betriebssystemfamilien kennenzulernen, scheint es daher lohnenswert, eine von Grund auf aspektorientiert entworfene und implementierte Betriebssystemfamilie zu erstellen. Ein solches “AspectOS” würde zahlreiche neue Anwendungen der in dieser Arbeit vorgestellten Werkzeuge und Ideen mit sich bringen. Daher erläutern die folgenden Abschnitte diese Vision etwas genauer.

10.3.1 Vision: AspectOS

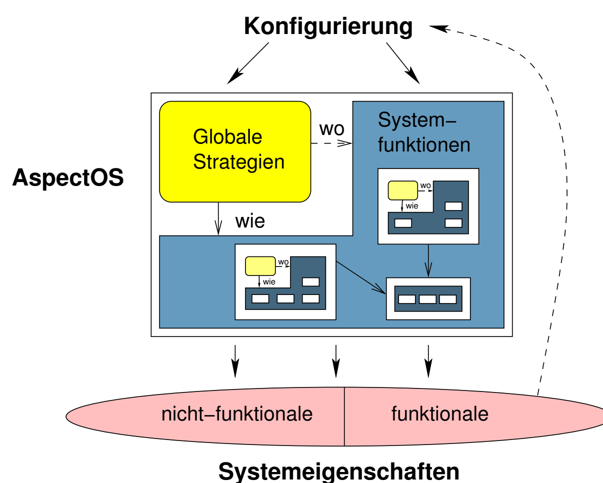


Abbildung 10.3: Das AspectOS Modell

Abbildung 10.3 zeigt das Modell des AspectOS. Dahinter steckt die Idee, dass Strategien von den eigentlichen Systemfunktionen und der funktionalen Hierarchie, die sie bilden, getrennt betrachtet werden. Dies ist ein bewährtes Entwurfsprinzip:

> “Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details.”
>
> Silberschatz/Galvin [98]

Leider ist dieses Prinzip ohne aspektorientierte Implementierungstechniken nur schwer umzusetzen. Der Grund besteht darin, dass hinter den meisten strategischen Entwurfsentscheidungen nicht nur ein “wie”, sondern auch noch ein “wo” steckt. Tabelle 10.1 auf der nächsten Seite gibt einige Beispiele für globale Strategien in Betriebssystemen an und erläutert, was das “wie” und das “wo” dafür jeweils bedeuten.

Strategie	**“wie” und “wo”**
Schutz	Wo erfolgt die Trennung von Systemkomponenten durch Adressräume oder Privilegebenen? Wie werden beispielsweise Adressräume implementiert (seitenbasiert/segmentbasiert)? Wo werden Authentifikationsinformationen im System abgelegt? Wie, das heißt auf Basis welcher Informationen, erfolgt eine Authentifikation?
Synchronisation	Wo erfolgt die Synchronisation mit Unterbrechungen, zwischen CPUs im Mehrprozessorbetrieb und zwischen Fäden beim Zugriff auf Betriebsmittel? Wird grob- oder feingranular gesperrt? Wie, das heißt mit welchen Mitteln, wird synchronisiert (Pro-Epilog-Modell/hardwaremäßiges Sperren von Unterbrechungen, klassische/optimistische Verfahren)?
Nebenläufigkeit	Wo sollen für bestimmte Aufgaben eigene Kontrollfäden erzeugt werden? Wie erfolgt die Abarbeitung von Systemfäden (kooperativ/präemptiv)?
Ressourcenvergabe	Wo können mehrere Fäden um ein Betriebsmittel konkurrieren (CPU Zuteilung, Geräte)? Wie soll die Zuteilung erfolgen (prioritätsbasiert/First-Come-First-Serve/andere Strategie)? Wo soll Verbrauchskontrolle (zum Beispiel Zeit, Energie) oder Buchführung durchgeführt werden? Wie soll der Ressourcenverbrauch ermittelt werden.
Kommunikation	Wo werden die Datenstrukturen zur Interprozesskommunikation platziert (im Prozesskontrollblock/separat)? Wie kommunizieren Fäden (Port/Mailbox/gemeinsamer Speicher)?
Pufferung	Wo werden Pufferspeicher eingerichtet (blockorientierte Geräte, Verzeichniseinträge von Dateisystemen)? Wie werden die eingerichteten Puffer implementiert?
Verteilung	Wo sollen in einem verteilten Familienmitglied die einzelnen Systemkomponenten platziert werden und welche davon sind zu replizieren? Wie soll die Kommunikation zwischen verteilten Systemkomponenten erfolgen?
Fehlertoleranz	Wo sollen etwa durch Redundanz Fehler toleriert werden? Wie, also mit welchen Verfahren, soll Fehlertoleranz erreicht werden?
Fehlerbehandlung	Wo sollen Fehlersituation erkannt werden? Wie ist zu reagieren (Ausnahme/Meldung)?
Beobachtung	Wo sollen für eine Systembeobachtung Daten gesammelt werden (Prozesszustände,Kontrollfluss)? Wie ist mit den gesammelten Daten zu verfahren?

Tabelle 10.1: Das “wie” und “wo” globaler Strategien im Betriebssystembau

Durch die Trennung von Strategien und Funktionen (im Sinne funktionaler Hierarchien) und mit Hilfe einer aspektorientierten Implementierung kann nun das "wo" der Strategien modular durch Aspekte umgesetzt werden. Es handelt sich dabei um ein *Crosscutting Concern* und durch die modulare Implementierung wird eine vergleichsweise einfache Konfigurierbarkeit erreicht. Das "wie" kann durch die Auswahl einer herkömmlichen Funktion implementiert werden.

Die Abbildung 10.3 zeigt außerdem, dass das hier angewendete Entwurfsprinzip der Trennung von Strategien und Funktionen nicht nur auf der globalen Ebene, sondern auch rekursiv innerhalb einzelner Funktionen angewendet werden kann und sollte. So hat etwa das Fallbeispiel zum abstrakten Prozessszustand gezeigt, dass es innerhalb der Funktion "Prozessverwaltung" lokale *Crosscutting Concerns* gibt.

Durch die Konfigurierung wird der Funktionsumfang sowie das "wo" und "wie" der Systemstrategien festgelegt. Dadurch ergibt sich ein spezielles Familienmitglied mit spezifischen Systemeigenschaften. Dies sind funktionale Eigenschaften wie die Unterstützung bestimmter Geräte, Dateisysteme, Kommunikationsprotokolle und Nutzerschnittstellen sowie auch nicht-funktionale Eigenschaften wie die Codegröße, Laufzeiten, Kommunikations- und Unterbrechungsbehandlungslatenzen, Robustheit, Energieverbrauch und Sicherheit. Ziel der Konfigurierung ist es, für eine bestimmte festgelegte Menge funktionaler Eigenschaften, die nicht-funktionalen Eigenschaften mit einer bestimmten Gewichtung und Richtung zu optimieren. Diese Optimierungsgrößen sind messbar oder zumindest abschätzbar, so dass wie in einem Regelkreis eine Rückkopplung zur Konfigurierung erfolgen kann. Dies ist durch einen entsprechenden Pfeil in Abbildung 10.3 gezeigt worden.

Interessante zukünftige Perspektiven könnten sich auch ergeben, wenn die Möglichkeit geschaffen würde, den hier skizzierten Regelkreis nicht nur durch einen Entwickler, der das System statisch mit anderen Konfigurierungseinstellungen neu generieren kann, steuern zu lassen, sondern durch ein Programm. Dieses könnte nach einem vorgegebenen Algorithmus die Rolle des Entwicklers übernehmen. Überlegenswert wäre auch die Veränderung von Strategien zur Laufzeit. Entsprechende Erweiterungen an AspectC++, die den dynamischen Austausch von Aspekten gestatten, wären denkbar. In diesem Fall könnten globale Systemstrategien zur Laufzeit an ein geändertes Anforderungsprofil angepasst werden. Dies wäre vermutlich bei eingebetteten Systemen weniger von Interesse als bei Spezialzweck-*Server*-Systemen, bei lang laufenden Systemen für Wartungs- und Diagnoseaufgaben oder bei offenen, mobilen Systemen.

Damit würde ein dynamisches AspectOS ähnliche Ziele verfolgen wie die schon vor mehreren Jahren vorgestellten reflektiven Betriebssysteme wie Apertos [117]. Allerdings ist dort modellbedingt jede Interaktion zwischen Objekten sehr kostspielig und erfordert selbst im optimierten Fall noch eine Überwachung [60]. Dagegen sollte beim AspectOS dem Familiengedanken folgend je nach Anwendungsszenario ein unterschiedlicher Grad an Dynamik gestattet werden. Wenn eine Gruppe alternativer, dynamisch austauschbarer Aspekte zum Beispiel vorab bekannt ist, können mit entsprechenden Erweiterung von AspectC++ mögliche

Verbindungspunkte bereits zum Konfigurierungszeitpunkt bestimmt und mittels minimaler Eingriffe vorbereitet werden, so dass ein Austausch von Aspekten zur Laufzeit ebenso wie deren Ausführung eine sehr viel kleinere Infrastruktur erfordert.

10.3.2 Erwartete Ergebnisse

Von der Entwicklung einer solchen aspektorientierten Betriebssystemfamilie kann man sich vor allem eine hochgradige Konfigurierbarkeit versprechen. Tabelle 10.1 hat gezeigt, wie weit die Konfigurbarkeit gehen kann. So kann im Bereich der Spezialzwecksysteme eine sehr gute anwendungsspezifische Anpassung erfolgen. Darüber hinaus erlaubt die Änderbarkeit das Experimentieren mit globalen Systemstrategien, was bisher nicht möglich war.

Interessant ist auch die Beobachtung, dass die in Tabelle 10.1 aufgeführten globalen Strategien alles einschließen, was die sogenannte Systemarchitektur ausmacht. Vergleicht man beispielsweise ein typisches Mikrokernsystem wie Windows NT [29] mit einem monolithischen System wie Linux [13], so sind die Unterschiede bezüglich funktionaler Eigenschaften marginal. Viele Nutzer würden die Unterschiede gar nicht bemerken, wenn auf beiden Systemen die gleiche grafische Benutzeroberfläche laufen würde. Stattdessen sind es die verfolgten globalen Strategien, die die Unterschiede ausmachen. Dazu zählt beispielsweise, ob für Gerätetreiber ein eigener Kontrollfluss erzeugt wird und ob Treiber im privilegierten oder unprivilegierten Arbeitsmodus des Prozessors ablaufen sollen. Es handelt sich dabei um Teilaspekte der Strategien Nebenläufigkeit und Schutz. Das Fallbeispiel zur Fadensynchronisation hat gezeigt, dass die Frage der Fadenerzeugung für Gerätetreiber und deren Kommunikation mit der Außenwelt durchaus durch Aspekte implementierbar ist, so dass von einer aspektorientierten Betriebssystemfamilie die Konfigurierbarkeit aller architekturrelevanten Systemstrategien zu erwarten ist. Das Ergebnis wäre **Architekturtransparenz**, das heißt die Unsichtbarkeit der Betriebssystemarchitektur im Quellcode der funktionalen Systemkomponenten.

Durch Architekturtransparenz könnten verschiedene Architekturen auf Basis derselben funktionalen Systemkomponenten verglichen werden. So könnte praktisch bestätigt werden, dass nachrichten- und prozedurorientierte Systemkerne unter Vernachlässigung anderer Systemstrategien wie Schutz äquivalent und automatisch ineinander überführbar sind, wie es schon vor über 20 Jahren aufgrund theoretischer Überlegungen behauptet wurde [71].

Der praktische Wert eines architekturtransparenten Systems besteht beispielsweise darin, dass Gerätetreiber, die vielfach den größten Teil des Programmcodes eines Betriebssystems ausmachen, architekturneutral entwickelt werden könnten. In einem System wie Linux, wo diese Eigenschaft nicht gegeben ist, beschäftigt eine architektonische Änderung, nämlich das feingranulare Sperren, die Entwickler seit mehreren Jahren. In einem architekturtransparenten AspectOS sollte eine solche Änderung nur wenige Tage in Anspruch nehmen.

10.4 Zusammenfassung

Dieses Kapitel hat gezeigt, dass die Arbeiten mit Aspekten im Umfeld von Betriebssystemfamilien viel Raum für weiterführende Forschungen durch den Autor und andere Wissenschaftler lassen. Dabei stehen auf der sprachlichen Seite Fragen der Semantik und des Schutzes im Vordergrund. Eine bessere Unterstützung des Entwicklungsprozesses sollte durch Werkzeuge, Entwurfsmuster und eine durchgängige Methode erreicht werden. Diese Punkte haben allerdings nur noch sehr wenig mit Betriebssystemen zu tun, so dass hier der Kontakt zu Softwaretechnikern intensiviert werden sollte.

Auf der Ebene der Anwendung der in dieser Arbeit vorgestellten Werkzeuge und Techniken könnte die Entwicklung einer aspektorientiert entworfenen und implementierten Betriebssystemfamilie nach Jahren der Stagnation im Betriebssystembereich [91, 42] vielleicht zu einem neuen Denken führen.

Kapitel 11

Rückblick und Diskussion

11.1 Geleistete Arbeiten, Erfahrungen und Pläne

Das Ziel dieser Arbeit bestand darin, erste Erfahrungen mit dem Einsatz der aspektorientierten Programmierung im Kontext von Betriebssystemfamilien zu sammeln. Der Grund für diese Zielsetzung bestand in der Beobachtung, dass viele Probleme, die der Entwurf solcher Systemfamilien mit sich bringt, auf *Crosscutting Concerns* zurückzuführen ist. Dabei handelt es sich um Aufgaben eines Softwaresystems, die mit herkömmlichen Entwurfs- und Implementierungstechniken nicht modular umgesetzt werden können. Stattdessen liegt der resultierende Programmcode über weite Teile des Gesamtsystems verstreut vor. Dieses Phänomen ist angesichts des Prinzips *Separation of Concerns* allein schon schlimm genug, doch es wird im Kontext konfigurierbarer Softwarefamilien noch potenziert. Hier sollen möglichst viele Systemeigenschaften, egal ob sie nun als *Crosscutting Concern* eingestuft werden oder nicht, konfigurierbar sein, falls dies im Interesse des Anwenders ist. Damit kommt es zu dem Problem, dass Code für die Implementierung einer bestimmten Aufgabe nicht nur verstreut ist, sondern je nach Systemkonfiguration an unterschiedlichen Stellen vorliegen muss. In der Praxis führte dies meist dazu, dass *Crosscutting Concerns* nicht konfigurierbar ausgelegt wurden. Im Kontext von Betriebssystemen gibt es sehr viele *Crosscutting Concerns*, wie Tabelle 10.1 auf Seite 176 gezeigt hat. Es handelt sich dabei um Aufgaben, deren Entwurf strategischen Charakter hat, wie das Schutzkonzept oder die Synchronisation beim Zugriff auf Betriebsmittel. In diesen Punkten auf Konfigurierbarkeit zu verzichten, bedeutet eine starke Einschränkung einer Betriebssystemfamilie.

Der in dieser Arbeit verfolgte Ansatz das Problem zu lösen, besteht in der Anwendung des Konzepts der aspektorientierten Programmierung. Dabei wird mit programmiersprachlichen Mitteln die modulare Implementierung von *Crosscutting Concerns* ermöglicht. Mit diesem noch vergleichsweise jungen Konzept bestand nun die Hoffnung, auch globale Strategien in Betriebssystemfamilien konfigurierbar implementieren zu können. Ob dies allerdings tatsächlich so funktioniert, war zu Beginn der Arbeiten keinesfalls klar. Schließlich gibt es unterschiedliche Ar-

ten von *Crosscutting Concerns*, die möglicherweise spezifische sprachliche Mittel erfordern oder auch überhaupt nicht modularisierbar sind. Um hier Antworten zu finden, sollten alle erforderlichen Schritte unternommen werden, um am Ende konkrete Fallstudien durchführen zu können. Als Studienobjekt bot sich dafür die Betriebssystemfamilie PURE an, an deren Entwicklung der Autor beteiligt war.

Das Haupthindernis für die Durchführung solcher Fallstudien bestand in dem Fehlen geeigneter aspektorientierter Programmiersprachen, die für Betriebssystementwicklungen in Frage kommen. Dies gilt insbesondere, da eine sehr wichtige Anwendungsdomäne für Betriebssystemfamilien kleinste eingebettete Systeme sind, wo extrem sparsam mit Ressourcen umgegangen werden muss. Gleichzeitig sollte PURE auch nicht in einer anderen Programmiersprache komplett neu implementiert werden. Ein weiteres Hindernis bestand in dem Mangel an Modellen zur Beschreibung der Beziehungen zwischen Aspekten und Komponentencode, die zu dem Modell der funktionalen Hierarchie passen, was dem Entwurf von PURE aber auch anderer Systemfamilien zugrunde liegt.

Zur Lösung dieser Probleme wurde eine ganze Aspektwebersuite konzipiert und implementiert, die aus den Aspektwebern COMA, SOSP und AspectC++ besteht. Mit COMA wurde die Möglichkeit geschaffen, die Klassenstruktur von Subsystemen durch ein separates Aspektprogramm zu bestimmen, so dass die Struktur im Sinne des verfolgten Ansatzes leicht in Abhängigkeit von Anwendungsanforderungen konfiguriert werden kann. Messungen haben gezeigt, dass so der Ressourcenverbrauch sehr gut mit den Anforderungen skaliert. Der Aspektweber SOSP erlaubt die modulare Implementierung einer Strategie zur Einbettung von Funktionen an der Aufrufstelle. Auch dies hat einen erheblichen Einfluss auf den Ressourcenverbrauch. Beide Aspektweber behandeln *Crosscutting Concerns*, die für die Entwicklung von Programmfamilien in der Sprache C++ sehr wichtig sind. Sie sind jedoch nicht betriebssystemspezifisch. Für solche Fälle wurde im Rahmen der Fallstudien AspectC++ eingesetzt. Dabei handelt es sich um eine aspektorientierte Vielzweckspracherweiterung für C++, die semantisch und syntaktisch an AspectJ, eine entsprechende Erweiterung für Java, angelehnt ist. Alle drei Aspektweber basieren auf Quellcodetransformationen, die mit Hilfe von PUMA, einem System zur Analyse und Manipulation von C++ Code, durchgeführt werden. Neben diesen Werkzeugen wurde zur Unterstützung des Entwurfsprozesses das Modell der *Concern*-Hierarchien entwickelt, die funktionale Hierarchien um *Crosscutting Concerns* und konfigurierbare Funktionen erweitern.

Auf Basis dieser Lösungen konnten nun drei Fallstudien durchgeführt werden, mit deren Hilfe das Modell der *Concern*-Hierarchien und die entwickelten Werkzeuge direkt erprobt werden konnten. Beides hat sich bewährt und es konnte darüber hinaus gezeigt werden, dass es tatsächlich möglich ist, globale Strategien in Betriebssystemcode wie die Strategie zur Unterbrechungssynchronisation und die zur Fadensynchronisation modular und damit leicht konfigurierbar umzusetzen. Durch die konsequent sparsame Codegenerierung bei der Übersetzung von AspectC++ nach C++ konnte dies sogar ohne zusätzlichen Ressourcenverbrauch erreicht werden.

Mit diesen Aussagen wurde ein wichtiges Ziel der Arbeit erreicht. Daneben ging es aber auch darum, Erfahrungen zu sammeln, die vielleicht wieder zu neuen Ideen führen. Eine dieser Erfahrungen ist, dass man als Entwickler einer Programmfamilie mit Aspekten viel mehr Freiheiten hat, das Programmverhalten zu beeinflussen. In einer in Form von Schichten schrittweise erweiterten Programmfamilie bedeutet das, dass eine später hinzugefügte Schicht in Lage ist, Ergänzungen in Form von *Advice*-Code oder auch Datenelementen in Basisschichten zu implementieren. Der Entwurf der Basisschichten einer Programmfamilie ist üblicherweise besonders schwierig, da versucht werden muss, keine Entwurfsentscheidungen zu treffen, die das Wachstum der Familie beeinträchtigen könnten. Dies ist durch die Mächtigkeit von Aspekten nun leichter. Andererseits hat der Entwickler nun ein Paradigma mehr zur Verfügung und die Qual der Wahl wird umso größer. Auf jeden Fall scheint es sinnvoll zu sein, Aspekte einzusetzen, wenn ein *Crosscutting Concern* mit Coderedundanz wie im Fall eines Kontrollflussverfolgungsaspekts zu implementieren ist. Ebenfalls sinnvoll ist ihre Anwendung bei konfigurierbaren *Crosscutting Concerns* mit unterschiedlichen Verbindungspunkten, beispielsweise dem "wo" globaler Systemstrategien. In anderen Fällen ist die Lage weniger klar. Soll zum Beispiel eine konfigurierbare Speicherverwaltung vorgesehen werden, wäre es eventuell sinnvoll, die Datenelemente des Prozesskontrollblocks, die der Speicherverwaltung dienen, mit Hilfe eines Aspekts von außen einzufügen, damit der zu konfigurierende Code zusammengehalten wird. Die Gefahr bei solchen vergleichsweise unbedeutenden Aspekten ist, dass der Entwickler irgendwann nicht mehr versteht, was in dem Programm tatsächlich an welcher Stelle passiert. Wie weit man mit Aspekten gehen sollte, müssen weitere Erfahrungen in der Zukunft zeigen. Ganz sicher hängt diese Frage aber auch von der gegebenen Unterstützung durch Entwicklungswerkzeuge ab. Wenn beispielsweise die Einflussnahme eines Aspekts der Speicherverwaltung auf den Prozesskontrollblock geeignet visualisiert würde, spräche wohl nichts gegen diese Implementierungsvariante.

Es gibt noch viele weitere Entwurfsfragen, die zum gegenwärtigen Zeitpunkt noch schwer zu beantworten sind. So stellt sich die Frage, wieviel Wissen Aspekte über den Komponentencode, auf den sie wirken, nutzen sollten, um nicht empfindlich auf Änderungen am Komponentencode zu reagieren. Vielleicht sollte zum Beispiel auf *Advice* für Attributzugriffe komplett verzichtet werden. Tatsächlich wurde diese AspectC++ Spracheigenschaft im Rahmen der Fallstudien nicht verwendet. Im Beispiel der Unterbrechungssynchronisation wurde sogar nur auf der Ebene von Subsystemen unter Verwendung weniger Klassennamen gearbeitet.

Eine weitere schwierige Entwurfsfrage bezieht sich auf Abhängigkeiten vom Aspektcode. Bei AspectC++ und AspectJ können durch gewöhnlichen Komponentencode Methoden von Klassen aufgerufen werden, die erst durch Einfügungen entstehen. Denkbar ist hier zum Beispiel die Erzeugung von Laufzeittypinformationen, die von einem Aspekt generiert und vom Komponentencode benutzt werden. Ist dies ein schlechter Entwurfsstil, weil hier Aspekte zur Codegenerierung missbraucht werden oder ist es völlig legitim, weil es sich um ein *Crosscutting Concern* handelt? Noch gehen die Meinungen in dieser Frage auseinander. Eine

wichtige Aufgabe für die Zukunft wird daher darin bestehen, weitere Erfahrungen zu sammeln und Entwurfsmuster für die aspektorientierte Entwicklung von Programmfamilien zu identifizieren. Dies könnte gerade für Einsteiger eine wichtige Hilfe sein, um gute Anwendungsfälle für Aspekte zu erkennen und gleichzeitig das Konzept nicht überzustrapazieren.

Die während der Arbeit gewonnenen Erkenntnisse führten zu der Idee, eine aspektorientierte Betriebssystemfamilie von Grund auf neu zu entwerfen. Der Grund dafür ist die Einsicht, dass Aspekte zwar prinzipiell auf beliebigen Komponentencode wirken können, doch ein "design for aspect intervention", das heißt eine weitsichtige Strukturierung des Komponentencodes in Hinblick auf Aspekte, vorteilhaft wäre, um bestimmte Aspekte zu implementieren. Das Ziel dabei wäre, durch eine konfigurierbare Implementierung wichtiger globaler Strategien und der Trennung dieser vom Komponentencode zu Architekturtransparenz zu kommen. Mit den in diesem Rahmen entwickelten Aspekten, die sicherlich zum Teil noch Erweiterungen auf der Sprachseite erfordern, wäre nicht nur dem Bereich der Betriebssystemfamilien gedient, sondern auch der Wart- und Erweiterbarkeit herkömmlicher Systeme. Entwurfsentscheidungen, die bisher strategischen Charakter hatten, würden nach diesem Plan ihren Schrecken verlieren, so dass Anpassungen globaler Strategien leicht, möglicherweise sogar interaktiv, machbar wären.

11.2 Vorgehen und Einordnung

Das Vorgehen im Rahmen dieser Arbeit war geprägt von dem Ziel, praktische Ergebnisse zu erzielen und zu Aussagen über den Nutzen von AOP für Betriebssystemfamilien zu kommen. Voraussetzung für diese Entwicklungen waren Werkzeuge, ein Entwurfsmodell und möglichst auch eine Entwurfsmethode. All diese Punkte sollten an sich von anderen darauf besser spezialisierten Gruppen bearbeitet werden. Leider ist jedoch so, dass beispielsweise derzeit niemand in der AOP Forschergemeinde ein Interesse daran hat, eine aspektorientierte Spracherweiterung für beispielsweise C++ zu entwicklen. Da bereits Sprachen wie AspectJ existieren, an denen Eigenschaften von Aspekten studiert werden können, halten viele den Aufwand für eine weitere Sprache – gerade, wenn sie so komplex wie C++ ist – für nicht gerechtfertigt.

So blieb in diesem Punkt nur die Eigeninitiative und auf der Implementierungsseite sind am Ende die geleisteten Arbeiten bei den Werkzeugen deutlich größer geworden als bei den Fallstudien. Natürlich wäre es auch möglich gewesen, den Weg von AspectC einzuschlagen, der aspektorientierten Spracherweiterung für C, bei der für Fallstudien ein Mensch als Übersetzer eingesetzt wurde. Bezüglich des in dieser Arbeit vertretenden Ansatzes ist es jedoch so, dass dieser auch für zukünftige Arbeiten noch soviel Potential erkennen lässt, dass eine nachhaltige Lösung bei den Werkzeugen unverzichtbar war.

Mit der Entwicklung einer Programmiersprache zum Ziel des Betriebssystembaus ist diese Arbeit nicht die erste ihrer Art. Die bekanntesten Beispiele sind die

Entwicklungen von C und Oberon [116]. Einerseits sitzt man mit einem solchen Beitrag immer ein wenig zwischen den Stühlen, doch gleichzeitig wird die wichtige Aufgabe übernommen, zumindest für einige Zeit eine Brücke zwischen der Welt der Sprachen und der Betriebssysteme zu bilden, mit deren Hilfe jede Gruppe von der anderen lernen kann. Die einen könnten offener gegenüber den Möglichkeiten moderner Programmiersprachen werden, während die anderen nicht C und C++ aus verschiedenen Gründen verdammen sollten, ohne eine akzeptable Alternative anzubieten.

Ein Vergleich der vorliegenden Arbeit mit ähnlichen Arbeiten zeigt, dass man hier sehr gut im Rennen liegt. Keine Arbeitsgruppe konnte bisher Fallstudien, wie sie hier präsentiert wurden, im Betriebssystemkontext vorweisen. Die entwickelte Sprache AspectC++ und der dafür im Internet verfügbare Übersetzer sorgen für eine sehr positive Resonanz. So gibt es bereits im August 2002 über 50 eingetragene Anwender mit schnell steigender Tendenz. Der größte Teil dieser Nutzer kommt aus Firmen, die in den Bereichen der eingebetteten oder mobilen Systeme tätig sind. Auf diese Weise sind bereits einige Kontakte auch zu sehr bekannten Konzernen zustande gekommen. Dies lässt den Schluss zu, dass den durchgeführten Arbeiten ein tatsächlicher Bedarf gegenübersteht.

Literaturverzeichnis

[1] A. V. Aho, R. Sethi und J. D. Ullman. *Compiler — Principles, Techniques, and Tools*. Addison-Wesley Publishing, 1986. ISBN 0-201-10194-7.

[2] M. Aksit, R. Mostert und B. Haverkort. Compiler Generation Based on Grammar Inheritance. Technical report, University of Twente, Netherlands, 1990.

[3] O. Aldawud, T. Elrad und A. Bader. A UML Profile for Aspect Oriented Modeling. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, USA, Okt. 2001.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Mai 1994. (DIKU report 94/19).

[5] S. Apel. Eine universelle graphische Entwicklungsumgebung für ausführbare Operationsnetzwerke. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Germany, 2002.

[6] AT&T. UNIX System V AT&T C++ Language System: Release 2.0, Library Manual, 1989. Select code 307-145.

[7] M. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[8] D. Batory und B. J. Geraci. Validating Component Compositions in Software System Generators. In M. Sitaraman, editor, *Fourth International Conference on Software Reuse*, Seiten 72–81, Orlando, Florida, 1996. IEEE Computer Society Press.

[9] D. Batory und B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, Feb. 1997.

[10] D. Batory und S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(1):355–398, Jan. 1992.

[11] L. Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conf. on Advanced Information Systems Engineering (CAISE-99), 6th Doctoral Consortium*, Heidelberg, Germany, Juni 1999.

[12] L. Baum, L. Geyer, G. Molter und S. R. P. Sturm. Architecture-Centric Software Development Based on Extended Design Spaces. In *Proceedings of the 2nd Int'l Workshop on Development and Evolution of Software Architectures for Product Families (ARES)*, Las Palmas de Gran Canaria, Spain, Feb. 1998.

[13] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus und D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1998. ISBN 0-2011-33143-8.

[14] C. Becker und K. Geihs. Quality of Service — Aspects of Distributed Programs. In *Proceedings of the ICSE'98 Workshop on Aspect-Oriented Programming*, 1998.

[15] D. Beuche, L. Büttner, D. Mahrenholz, F. Schön und W. Schröder-Preikschat. jPure — A Purified Java Execution Environment for Controler Networks. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES 2000)*, Paderborn, Okt. 2000.

[16] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, Mai 1999.

[17] D. Beuche, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. Streamlining Object-Oriented Software for Deeply Embedded Applications. In *Proceedings of the TOOLS Europe 2000*, Mont-Saint-Michel, France, Juni 2000.

[18] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan und D. Bohman. Microkernel Operating System Architecture and Mach. *Journal of Information Processing*, 14(4), März 1992.

[19] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas und B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings. OONSKI '94*, Oregon, USA, Apr. 1994.

[20] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4), Aug. 1986.

[21] G. Bracha und W. Cook. Mixin-based Inheritance. In *Proceedings of the ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, Seiten 303–311, Okt. 1990.

[22] S. Chiba. Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 285–299, Okt. 1995.

[23] S. Clarke und R. J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Canada, Mai 2001.

[24] Y. Coady, A. Brodsky, D. Brodsky, J. Pomkoski, S. Gudmundson, J. S. Ong und G. Kiczales. Can AOP Support Extensibility in Client-Server Architectures? In *European Conference on Object-Oriented Programming (ECOOP), Aspect-Oriented Programming Workshop*, Juni 2001.

[25] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson und J. S. Ong. Structuring Operating System Aspects. *Communications of the ACM*, Seiten 79–82, Okt. 2001.

[26] Y. Coady, G. Kiczales, M. Feeley und G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Internation Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[27] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault und E.-N. Volanschi. Tempo: Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3es), 1998.

[28] P. Courtois, F. Heymans und D.L.Parnas. Concurrent Control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.

[29] H. Custer. *Inside WINDOWS-NT*. Microsoft Press, 1993.

[30] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Dissertation, Technische Universität Ilmenau, Germany, 1998.

[31] K. Czarnecki, L. Dominick und U. W. Eisenecker. Aspektorientierte Programmierung mit C++, Teil 1–3. *iX Magazin*, 8–10, 2001.

[32] K. Czarnecki und U. Eisenecker. Synthesizing Objects. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, Seiten 18–42, Lisbon, Portugal, 1999. Springer Verlag.

[33] K. Czarnecki und U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley Publishing, 2000.

[34] S. Dangeti, T. Ramasamy und J. Murugan. Runtime Weaving of Aspects using Dynamic Code Instrumentation Technique for Building Adaptive Software Systems. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Seiten 12–15, Enschede, The Netherlands, Apr. 2002.

[35] E. Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, Seiten 341–346, Mai 1968.

[36] R. Douence, P. Fradet und M. Südholt. Detection and Resolution of Aspect Interactions. Technical Report No. 4435, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes, France, Apr. 2002.

[37] B. P. Douglass. *Real-Time UML*. Addison-Wesley, 2000. ISBN 0-201-65784-8.

[38] S. A. Edwards. *Languages for DIGITAL Embedded Systems*. Kluwer Academic Publishers, 2000. ISBN 0-7923-7925-X.

[39] T. Elrad, R. E. Filman und A. Bader. Aspect-oriented Programming. *Communications of the ACM*, Seiten 29–32, Okt. 2001.

[40] M. Friedrich, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. Efficient Object-Oriented Software with Design Patterns. In K. Czarnecki und U. W. Eisenecker, editors, *Generative and Component-Based Software-Engineering. First International Symposium, GCSE'99*, Erfurt, Germany, Sep. 1999. Springer-Verlag. Revised Papers. LNCS 1799.

[41] A. A. M. Fröhlich. *Application-Oriented Operating Systems*. Dissertation, Technische Universität Berlin, 2001.

[42] A. A. M. Fröhlich und W. Schröder-Preikschat. Operating Systems: are we finally ready to move forward after 30 years of stagnation? In *Proceedings of the Wild and Crazy Ideas Session of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, USA, Nov. 2000.

[43] A. Gal. Reconciliation of an Object-Oriented Runtime Environment and Resource Restricted Systems. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Germany, 2001.

[44] A. Gal, W. Schröder-Preikschat und O. Spinczyk. On Minimal Overhead Operating Systems and Aspect-Oriented Programming. In *Proceedings of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, Budapest, Hungary, Juni 2001. ISBN 84-699-5329-X.

[45] A. Gal, W. Schröder-Preikschat und O. Spinczyk. Open Components. In *D. H. Lorenz, V. C. Sreedhar (Eds.): Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa, USA, Okt. 2001. Northeastern University Tech. Report NU-CCS-01-06.

[46] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[47] L. Goldschlager und A. Lister. *Informatik — Eine moderne Einführung*. Hanser Verlag, 1990. ISBN 3-446-15766-2.

[48] M. L. Griss. Implementing Product-Line Features by Composing Component Aspects. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, Seiten 271–288, 2000.

[49] U. Haack. Implementierung und Bewertung von Sperrmechanismen für multiprozessorfähige, parallele Echtzeitbetriebssystemkerne am Beispiel von PEACE. Diplomarbeit, Technische Universität Berlin, Okt. 1996.

[50] A. N. Habermann, L. Flon und L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[51] J. Hannemann und G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2002)*, Seattle, Washington, USA, Nov. 2002.

[52] W. Harison und H. Ossher. Subject-Oriented Programming (a Critique on Pure Objects). In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Seiten 411–428, Woshington, D.C., Sep. 1993. ACM.

[53] F. J. Hauck, U. Becker, M. Geier, E. M. U. Rastofer und M. Steckermeier. AspectIX: a quality-aware, object-based middleware architecture. In *Proceedings of the 3rd IFIP Int. Conf. on Distrib. Appl. and Interoperable Sys. - DAIS*, Krakow, Poland, Sep. 2001. Kluver.

[54] G. Hjálmtýsson und B. Gray. Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program. In *Proceedings of the 1998 USENIX Technical conference*, 1998.

[55] C. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[56] C. M. Hoffmann und M. J. O'Donnell. Pattern Matching in Trees. *JACM*, 29(1):68–95, 1982.

[57] M. Hohmuth, H. Tews und S. G. Stevens. Applying source-code verification to a microkernel — The VFiasco project. Technical report, Technische Universität Dresden, Fakultät Informatik, 2002.

[58] International Organization for Standardization. Standard ISO/IEC 14882:1998(E), Programming Languages - C++, First Edition, 1998.

[59] International Organization for Standardization. Standard ISO/IEC 9899:1999(E), Programming Languages - C, Second Edition, 1999.

[60] J. Itoh, R. Lea und Y. Yokote. Using meta-objects to support optimisation in the Apertos operating system. In *USENIX Conference on Object Oriented Technologies (COOTS)*, Juni 1995.

[61] S. C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, Seiten 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[62] K. C. Kang, S. G. Cohen, J. A. Hess, W. Novak und A. S. Peterson. Feature-Oriented Domain Analysis Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Nov. 1990.

[63] K. C. Kang, S. Kim, J. Lee, K. Kim und E. Shin. FORM : A Featured-Oriented Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5, 1998.

[64] B. W. Kernighan und D. M. Ritchie. *The C Programming Language, First Edition*. Prentice Hall, Feb. 1978.

[65] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm und W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, Seiten 59–65, Okt. 2001.

[66] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm und W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, Juni 2001.

[67] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier und J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, Feb. 1997.

[68] T. Kishi und N. Noda. Aspect-Oriented Analysis for Product Line Architecture. In *OOPSLA 2000 workshop on Advanced Separation of Concerns*, Okt. 2000.

[69] W. P. Kowalk. *Korrekte Software: Semantik, Spezifikation, Verifikation und Testen von Programmen*. BI-Wis. Verl., 1993. ISBN 3-411-16001-2.

[70] T. G. Lane. Studying Software Architecture Through Design Spaces and Rules. Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University, Nov. 1990.

[71] H. C. Lauer und R. M. Needham. On the Duality of Operating System Structures. *ACM Operating Systems Review*, 13(2):3–19, Apr. 1979.

[72] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing, 1995. ISBN 0-201-11972-2.

[73] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[74] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), Mai 1988.

[75] C. V. Lopes und G. Kiczales. D: A Language Framework for Distributed Computing. Technical Report SPL97-010 P9710047, Xerox PARC, Feb. 1997.

[76] D. Mahrenholz. Minimal Invasive Monitoring. In *Proceedings of the Fourtieth IEEE International Symposium in Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, Mai 2001.

[77] D. Mahrenholz, O. Spinczyk, A. Gal und W. Schröder-Preikschat. An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family. In *Proceedings of the 5th ECOOP Workshop on Object-Orientation and Operating Systems*, Seiten 49–54, Malaga, Spain, Juni 2002. SERVITEC. ISBN: 84-699-8733-X.

[78] S. McCanne und V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, Seiten 259–270, Dez. 1993.

[79] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller und R. Marlet. Specialization Tools and Techniques for Systematic Optimization of System Software. *ACM Transactions on Computer Systems*, 19(2):217–251, Mai 2001.

[80] H.-P. Messmer. *PC-Hardwarebuch: Aufbau, Funktionsweise, Programmierung; ein Handbuch nicht nur für Profis*. Addison-Wesley, 1995. ISBN 3-89319-710-9.

[81] S. Microsystems. Remote Procedure Call, Version 2. RFC 1057, Juni 1988.

[82] G. Muller, E. Volanschi und R. Marlet. Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Seiten 116–125, Amsterdam, The Netherlands, Juni 1997. ACM Press.

[83] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai und M. A. Kersten. Does Aspect-oriented Programming Work. *Communications of the ACM*, Seiten 75–77, Okt. 2001.

[84] P. Netinant, C. A. Constantinides, A. Bader und T. Elrad. Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. In *International Conference on Parallel and Distributed Techniques and Applications (PDPTA 2000)*, Las Vegas, Nevada, Juni 2000. special session on Aspect-Oriented Programming.

[85] P. Netinant, C. A. Constantinides, T. Elrad und M. E. Fayad. Supporting Aspectual Decomposition in the Design of Operating Systems. In *Proceeding of the 3rd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2000)*, Seiten 38–46. Universidad de Oviedo, Juni 2000. ISBN 84-8317-222-4.

[86] J. Nolte. *Duale Objekte — Ein Modell zur objektorientierten Konstruktion von Programmfamilien für massiv parallele Systeme*. Dissertation, Technische Universität Berlin, 1994.

[87] Object Management Group. OMG Unified Modeling Language Specification, Version 1.4, Sep. 2001. http://www.omg.org/uml.

[88] H. Ossher und P. Tarr. Using Multidimensional Separation of Cconerns to (Re)shape Evolving Software. *Communications of the ACM*, Seiten 43–50, Okt. 2001.

[89] D. L. Parnas. Some Hypothesis About the Uses Hierarchy for Operating Systems. Technical report, Technische Hochschule Darmstadt, Fachbereich Informatik, 1976.

[90] T. J. Parr. *Language Translation Using PCCTS and C++*. Automata Publishing Company, 1993. ISBN 0-9627488-5-4.

[91] R. Pike. Systems Software Research is Irrelevant, Feb. 2000. CS Colloquium, Columbia, http://cm.bell-labs.com/who/rob/utah2000.pdf.

[92] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole und K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP'95)*, Colorado, USA, Dez. 1995.

[93] C. Pu, H. Marsalin und J. Ioannides. The Synthesis Kernel. *Computing Systems*, 1(1), 1988.

[94] E. Pulvermüller, H. Klaeren und A. Speck. Aspects in Distributed Environments. In K. Czarnecki und U. W. Eisenecker, editors, *Generative and Component-Based Software-Engineering. First International Symposium, GCSE'99*, Erfurt, Germany, Sep. 1999. Springer-Verlag. Revised Papers. LNCS 1799.

[95] F. Schön, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998.

[96] F. Schön, W. Schröder-Preikschat, O. Spinczyk und U. Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, Seiten 270–277, Newport Beach, California, März 2000. IEEE Computer Society. ISBN 0-7695-0607-0.

[97] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[98] A. Silberschatz und P. Calvin. *Operating System Concepts*. Addison-Wesley, 1994. ISBN 0-201-59292-4.

[99] Y. Smaragdakis und D. Batory. Implementing Layered Design with Mixin Layers. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, Seiten 550–570, Brussels, Belgium, 1998.

[100] W. Stallings. *Operating Systems*. Prentice-Hall, 1995. ISBN 0-13-180977-6.

[101] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[102] Sun Microsystems. The Java 2 Platform, Standard Edition, v 1.4.0, API Specification, 2002. http://java.sun.com/docs.

[103] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.

[104] C. Szyperski und C. Pfister. Workshop on Component-Oriented Programming. In *ECOOP 1996 Workshop Reader*, 1997.

[105] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. ISBN 0-13-595752-4.

[106] The Embedded C++ Technical Committee. The Embedded C++ Specification, Version WP-AM-003, 1999.

[107] J. Turley. Embedded Processors. *PC Magazine*, Jan. 2002.

[108] E. Unruh. Prime Number Computation. ANSI X3J16-94-0075/SO WG21-462.

[109] M. Urban. The PUMA User's Manual, 2000. http://ivs.cs.uni-magdeburg.de/~puma.

[110] M. Urban. Ein Codeanalysesystem für die Familie der C-basierten Sprachen. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Germany, 2002.

[111] T. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, Mai 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[112] C. von Flach, G. Chavez und C. J. P. de Lucena. Design-level Support for Aspect-Oriented Software Development. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, USA, Okt. 2001.

[113] R. J. Walker, E. L. A. Baniassad und G. C. Murphy. An Initial Assessment of Aspect-oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering*, Mai 1999.

[114] E. D. Willink. *Meta-Compilation for C++*. PhD thesis, Computer Science Research Group, University of Surrey, Juni 2001.

[115] E. D. Willink und V. B. Muchnick. An Object-Oriented Preprocessor Fit for C++. *IEE Proceedings - Software*, 147(2):49–58, Apr. 2000.

[116] N. Wirth und J. Gutknecht. *Project Oberon — The Design of an Operating System and Compiler*. Addison-Wesley, 1993. ISBN 0-201-54428-8.

[117] Y. Yokote. The Apertos Reflective Operating System: The Concept and its implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992*, Seiten 414–434. ACM Press, Okt. 1992.

[118] X.-X. S. Zhang. *Practical Pointer Aliasing Analysis for C*. PhD thesis, Rutgers University, 1998.